Oracles and Confluence — A Fixpoint Semantics for Concurrent Constraint Programming

Sven-Olof Nyström
Box 311, S-751 05 Uppsala, Sweden
Phone: +48–18–18 25 00
Fax: +46–18–51 19 25

Abstract

We consider the problem of giving a fixpoint semantics for a parallel and non-deterministic programming language with fairness and arbitrary recursion, in which both finite and infinite computations are taken into account.

We first define an operational semantics based on oracles (finite and infinite strings over a countable alphabet), in which the non-deterministic choices are determined by the oracle. The oracle-based operational semantics associates an oracle with each computation, conversely there is for each oracle a set (possibly empty) of computations. We show that the oracle semantics satisfies certain confluence properties, which are to be later used in the correctness proof of the fixpoint semantics.

We give a fixpoint semantics for the language in which the semantics of an agent is given as a function from oracles to a pair consisting of a closure operator over the domain of constraints and a set of conditions which describe when it is legal to select a particular branch. It is shown that this semantic model is sufficient to precisely determine the set of fair traces of an agent. The fixpoint semantics is, however, not fully abstract since the different behaviours of an agent are related to different oracles in a way that has nothing to do with the external behaviour of the agent. Also, the oracle-based fixpoint semantics dissects the behaviour of an agent into different branches in a way that depends on the syntactic structure of the program.

Finally we give a category-theoretic fixpoint semantics. This semantics is obtained from the oracle-based fixpoint semantic by removing information about the relationship between behaviour and oracles.

1 Introduction

Concurrent constraint programming (ccp) [11Γ19] can be seen as a generalised model of asynchronous concurrent computation. In ccp the processes record information in a store. No information is ever removed from the store and the store will thus grow monotonously throughout the computation. The result of a computation
can be seen as the total amount of information that is added to the store. The two communication primitives are *ask* (check if a constraint is entailed by the store) and *tell* (add a constraint to the store).

In this paper we consider the problem of giving a fixpoint semantics for concurrent constraint programming which is able to describe both finite and infinite computations. It is known that it is not possible to give a fully abstract fixpoint semantics for a non-deterministic language if one wants to take into account infinite computations\(^1\) so the best one can hope for is a fixpoint semantics together with a simple abstraction operator. One would also like the fixpoint semantics to satisfy algebraic properties of various constructs in the programming language.

The approach in this paper is to base the semantics on *oracles*. An oracle is a sequence of integers representing the non-deterministic choices made by the agent. Now it is easy to see that many non-deterministic choices can only be made under certain conditions. This implies that we must find a way to record these conditions. The approach taken here is to record the conditions in the form of a *window* which is a set describing the final outcome of a computation. So one component of the semantics of an agent for a given oracle is a set of conditions i.e. a window. Since an agent with a given oracle is essentially deterministic it follows that we can use the techniques described by Saraswat, Rinard and Panangaden [19] and give computational behaviour as a closure operator (a function over constraints which satisfies some additional properties). The fixpoint semantics becomes quite simple even though both fairness and infinite computations are taken into account.

## 1.1 Related Work

Saraswat, Rinard and Panangaden [19] give fully abstract fixpoint semantics for various types of concurrent constraint programming languages. One of the languages under consideration is a non-deterministic language which allows arbitrary recursion. However only finite computations are considered.

The use of oracles to give the semantics of a non-deterministic concurrent language has been considered by a number of authors. For example Cadiou and Levy [3] gave the operational semantics of a parallel imperative language in which the scheduling of processes was determined by an oracle. Milner [12] gave an operational model of a non-deterministic language in which oracles were used to determine non-deterministic choices. Keller [7], Kearney and Staples [6] and Russell [18] have presented fixpoint semantics of various non-deterministic languages in which choices are determined by an oracle. However in all these models the choice was assumed to be independent of input i.e the languages in question do not allow the definition of a merge operator which is fair when the incoming data is finite.

## 1.2 An example

If we consider an agent together with a given oracle the oracle determines the non-deterministic choices made by the agent. Thus the resulting computation is

\(^1\)Apt and Plotkin [2], and Abramsky [1] presented results which both imply that a continuous fully abstract fixpoint semantics is not possible for concurrent constraint programming. The author [14] showed that it is not possible to give a fully abstract fixpoint semantics for a programming language which allows recursion, non-determinism and infinite output streams even if the semantics is not required to be continuous.
essentially deterministic and can be seen as a closure operator over the domain of constraints in the manner described by Saraswat et al. [19].

However this is not sufficient. Consider an agent

\[(X = 1 \Rightarrow Z = 3 \mid Y = 2 \Rightarrow W = 5),\]

The agent cannot select the first branch unless we know that the constraint \(X = 1\) will hold eventually. Similarly the second branch can only be selected if we know that \(Y = 2\) will hold eventually. It is also possible for the agent to be suspended without ever selecting a branch. This behaviour is acceptable only if we know that none of the two constraints will ever hold. It follows that it is necessary to include information in the semantic model describing when it is legal to select a certain branch. This information is given in the form of a \textit{window}.

The window gives one set of condition which must hold \textit{eventually} and one set of conditions which may \textit{never} hold. Conditions of the first type are of the form “this constraint must be entailed by the store” and conditions of the second type are of the form “this constraint must never become be entailed by the store”.

For example for the agent

\[(X = 1 \Rightarrow Z = 3 \mid Y = 2 \Rightarrow W = 5)\]

we have three branches where functionality and window are as follows.

1. For the first branch functionality is a function that adds the constraint \(Z = 3\) to the store provided that the store entails \(X = 1\). Window is the two conditions \(X = 1\) and \(Z = 3\) must hold eventually.

2. The second branch is similar.

3. For the third branch functionality is the identity function and the window is the condition that neither \(X = 1\) nor \(Y = 2\) may ever be entailed by the store.

In general the behaviour of an agent can be seen as a (continuous) function from oracles to functionality-window pairs.

1.3 Summary

The paper is organised as follows. Section 2 gives some basic definitions of mathematical concepts. Section 3 gives an axiomatic definition of constraint systems. In Section 4 we define a concurrent constraint programming language and give its operational semantics. The operational semantics is based on oracles in the manner discussed above. Since we consider infinite computations it is also necessary to define give a formal definition of fairness. In Section 5 we define an initial/final value semantics which describes the behaviour of an agent when it does not interact with other agents. We also give a semantics using traces. In Section 6 we show two confluence theorems the first is similar to the Church-Rosser theorem and the second theorem is a generalised version involving arbitrary (countably infinite) sets of computations. In Section 7 an oracle-based fixpoint semantics is given. In Section 8 the fixpoint semantics is shown to give precisely the same set of traces as the operational semantics. In Section 9 we consider a category-theoretic semantics.
This semantics is close to the oracle-based fixpoint semantics but the information regarding oracles has been hidden and the semantics is thus more abstract. The relation between the two fixpoint semantics is examined in Section 10.

2 Mathematical Preliminaries

In this section we give some definitions of mathematical concepts which we will need in the development of the semantic domain.

2.1 Partial orders, lattices and cpos

A pre-order is a binary relation $\leq$ which is transitive and reflexive. Given a pre-order $\leq$ over a set $L$ an upper bound of a set $X \subseteq L$ is an element $x \in L$ such that $y \leq x$ for all $y \in X$. The least upper bound of a set $X$ written $\bigvee \! X$ is an upper bound $x$ of $X$ such that for any upper bound $y$ of $X$ we have $x \leq y$. The concepts lower bound and greatest lower bound are defined dually. A function $f$ over a pre-order is monotone if $x \leq y$ implies $f(x) \leq f(y)$.

A partial order $(P, \leq)$ is a pre-order which is also antisymmetric. A non-empty set $R \subseteq P$ is directed if every finite subset of $R$ has an upper bound in $R$. For a set $S \subseteq P$ we will use the notation $S^u$ for the set

$$S^u = \{ x \in P \mid x \geq y, \text{ for } y \in S \}.$$  

A lattice is a partial order $(L, \leq)$ such that every finite subset has a least upper bound and a greatest lower bound. A complete lattice is a partial order $(L, \leq)$ such that every subset has a least upper bound (this implies that every subset also has a greatest lower bound). A function $f$ over a complete lattice $L$ is continuous if for every directed set $R \subseteq L$ we have $\bigvee \{ f(x) \mid x \in R \} = f(\bigvee R)$. For a complete lattice $L$ an element $x \in L$ is finite if for every directed set $R$ such that $x \leq \bigvee R$ there is some $y \in R$ such that $x \leq y$. For a lattice $L$ let $\mathcal{L}(L)$ be the set of finite elements of $L$. A lattice $L$ is algebraic if $x = \bigvee \{ y \in \mathcal{L}(L) \mid y \leq x \}$ for all $x \in L$ i.e. all elements of $L$ are either finite or the limit of a set of finite elements. Note that given an algebraic lattice $(L, \leq)$ and a monotone function $f$ over $\mathcal{L}(L)$ we can easily extend $f$ to a continuous function $f'$ over $L$ with $f'(x) = f(x) \Gamma$ for $x \in \mathcal{L}(L)$ and $f'(x) = \bigvee \{ f(y) \mid y \in \mathcal{L}(L), y \leq x \} \Gamma$ for $x \in L \setminus \mathcal{L}(L)$. In the following text algebraic lattices will be referred to as domains.

A cpo $(D, \subseteq)$ is a partial order with a least element where each directed set $R \subseteq D$ has a least upper bound in $D$.

3 Constraint Systems

To define a semantics for concurrent constraint programming we need a method to find a complete structure that incorporates a given constraint system and where operations like conjunction and existential quantification are well-defined and continuous. We accomplish this by assuming a set of formulas closed under the usual operations and an interpretation that gives the truth values of formulas given an assignment of values to (free) variables. Given this we use ideal completion to derive the desired domain. The resulting structure satisfies all axioms of cylindric algebra [4] that do not involve negation.
In contrast, Saraswat et al. [19] choose an axiomatic approach based on axioms from cylindric algebra and techniques from Scott’s information systems [20] to specify the properties of a constraint system.

There is some similarity between our approach and Kwiatkowska’s construction [8] which also uses ideal completion to obtain a constraint system that is closed under infinite limits.

**Definition 3.1** A pre-constraint system is a tuple \( \langle F, \text{Var}, \models, C \rangle \Gamma \) where \( F \) is a countable set of formulas, \( \text{Var} \) is an infinite set of variables, \( C \) is an arbitrary set, and \( \models \subseteq \text{Val} \times \bar{F} \) is a truth assignment where \( \text{Val} \) is the set of assignments \( \Gamma \) functions from \( \text{Var} \) to \( C \).

The only assumption we make about the structure of \( F \) is the following. If \( X \) and \( Y \) are variables and \( \phi \) and \( \psi \) are members of \( \Gamma \), the following formulas should also be members of \( F \):

- \( X = Y \)
- \( \exists X. \phi \)
- \( \phi \land \psi \)

The truth assignment \( \models \) should satisfy the following for each assignment \( V \Gamma \) formulas \( \phi \) and \( \psi \) and variables \( X \) and \( Y \).

1. \( V \models X = Y \) iff \( V(X) = V(Y) \).
2. \( V \models \exists X. \phi \) iff \( V' \models \phi \) for some assignment \( V' \) such that \( V(X') = V'(X') \) whenever \( X \neq X' \).
3. \( V \models \phi \land \psi \) iff \( V \models \phi \) and \( V \models \psi \).

We define a preorder \( \preceq \) between formulas by \( \phi \preceq \psi \) iff for any \( V \in \text{Val} \) such that \( V \models \phi \), we have \( V \models \psi \). Intuitively one can think of \( \phi \preceq \psi \) as meaning that \( \phi \) is weaker than \( \psi \) or that \( \psi \) implies \( \phi \). This gives immediately an equivalence relation \( \phi \equiv \psi \) defined by \( \phi \preceq \psi \) and \( \psi \preceq \phi \).

The general definition allows very powerful constraint systems where the basic operations can be computationally expensive or even uncomputable. However, in concurrent constraint programming one usually settles for much simpler structures. The following construction, sometimes referred to as the “term model,” gives us a concurrent constraint language with power and expressiveness comparable to concurrent logic languages such as GHC and Parlog.

**Example 3.2** Suppose we have a set of constant symbols \( \{a, \ldots\} \) and function symbols \( \{f, \ldots\} \). Let \( C \) be the smallest set that satisfies the following.

1. \( a \in C \) for any constant symbol \( a \).
2. \( f(E_1, \ldots, E_n) \in C \) if \( f \) is an \( n \)-ary function symbol and \( E_1, \ldots, E_n \in C \).

The formulas in \( F \) are simply formulas of the form \( E_1 = E_2 \) where \( E_1, E_2 \in C \).

To judge whether \( V \models E_1 = E_2 \) holds for an assignment \( V \) and expressions \( E_1 \) and \( E_2 \) simply replace each variable \( X \) in \( E_1 \) and \( E_2 \) with the corresponding value given by \( V(X) \). Then \( V \models E_1 = E_2 \) holds if and only if the resulting expressions are syntactically equal.
We would like to transform the preorder of formulas into a domain where equivalent formulas are identified and elements are added to make sure that each increasing chain has a limit. This is fairly straight-forward to accomplish using an ideal completion as follows.

**Definition 3.3** A *constraint* is a non-empty set $c$ of formulas such that

1. if $\phi \in c$ and $\psi \not\preceq \phi$ then $\psi \in c$
2. if $\phi, \psi \in c$ then $\phi \wedge \psi \in c$.

For a formula $\phi$ let $[\phi] = \{ \psi \mid \psi \preceq \phi \}$. Clearly $[\phi]$ is a constraint. A constraint $c$ for which there is a formula $\phi$ such that $[\phi] = c$ is a *finite constraint*. If we have a directed set $R$ of constraints then it follows from the definition of constraints that $\bigcup R$ is also a constraint. Let $\mathcal{U}$ be the set of constraints $\Gamma$ and $\mathcal{K}(\mathcal{U})$ the set of finite constraints.

The set of constraints form a complete lattice under the $\subseteq$ ordering with least element $\bot$ being the set of all formulas which hold under all assignments that is $\Gamma\{X = X, Y = Y, \ldots\}$. We will use the usual relation symbol $\subseteq$ for inclusion between constraints so that $c \subseteq d$ if and only if $c \subseteq d\Gamma$ and the symbol $\sqcup$ for least upper bound. As we noted previously if $R$ is a directed set of constraints it follows that $\bigcup R$ is a constraint and thus $\bigcup R = \bigcup R$. Note that for all formulas $\phi$ and $\psi\Gamma[\phi] \sqcup [\psi] = [\phi \wedge \psi]$. We can thus see least upper bound as an extension of conjunction. Also note that each constraint is either finite or a limit of a directed set of finite constraints which implies that the constraints form a domain.

We define for all variables $X$ a function $\exists_X : \mathcal{U} \to \mathcal{U}$ according to the following rules.

1. $\exists_X([\phi]) = [\exists_X \phi]\Gamma$ for formulas $\phi$.
2. $\exists_X(\sqcup R) = \sqcup_{\Gamma \in R} \exists_X(\Gamma)$ for directed sets $R \subseteq \mathcal{K}(\mathcal{U})$.

It is straightforward to prove that the function $\exists_X$ is well-defined and continuous.

**Remark** The fact that the $\exists_X$ function is continuous may seem counter-intuitive since in the arithmetic of natural numbers there is no $X$ such that all formulas $X > 0 \Gamma X > 1 \Gamma X > 2 \Gamma \ldots$ hold while $\exists X.X > n$ is true for any natural number $n$ and thus the set of formulas $\exists X.X > 0 \Gamma \exists X.X > 1 \Gamma \exists X.X > 2 \Gamma \ldots$ are always satisfied in any reasonable model for the natural numbers. This seems to imply that the existential quantifier is non-continuous. However when we perform ideal completion we add new infinite constraints. These are infinite sets which are not identified with their infinite conjunction so the resulting constraint system consists of finite constraints corresponding to the formulas mentioned above and infinite constraints corresponding to limits of directed sets of formulas. If the formulas of our language are inequalities such as the ones mentioned above and we have one variable $X\Gamma$ the resulting constraint system will contain the elements in the
This is of course closely related to the construction of non-standard models of arithmetic.

Example 3.4 Consider the term model mentioned in a previous example. The ideal completion gives us a new structure that is quite similar to the one we had previously except that we now can find a constraint \( c \) such that \( c \) holds if and only if all of the formulas

\[
\exists_Y(X = f(Y)), \quad \exists_Y(X = f(f(Y))), \quad \exists_Y(X = f(f(f(Y))))\ldots
\]

hold. □

3.1 From formulas to constraints

Given an assignment \( V \) and a constraint \( d \Gamma \) write \( V \models c \) to indicate that \( V \models \phi \) for all formulas \( \phi \in c \).

Clearly for constraints \( c \) and \( d \) such that \( c \sqsubseteq d \) we have \( V \models c \) whenever \( V \models d \). Also the following rules hold.

1. \( V \models [\phi] \) iff \( V \models \phi \Gamma \) for formulas \( \phi \).
2. \( V \models \exists_X c \) iff \( V' \models d \Gamma \) for an assignment \( V' \) such that \( V(Y) = V'(Y) \Gamma \) for variables \( Y \) distinct from \( X \).
3. \( V \models c \land d \) iff \( V \models c \) and \( V \models d \).

3.2 Properties of the constraint system

In the previous text we showed how a domain of constraints could be derived from a pre-constraint system. It should not come as a surprise that the operations defined over the domain of constraints (existential quantification, equality, and least upper bound i.e. conjunction) satisfy a number of algebraic properties. These properties correspond largely to the axioms of cylindric algebra [4]. Saraswat et al. [19] use the axioms of cylindric algebra to give an axiomatic definition of constraint systems.

Proposition 3.5 Suppose we have a pre-constraint system \( \langle F, \text{Var}, \models, C \rangle \Gamma \) and that \( \langle U, \sqsubseteq \rangle \) is the corresponding domain of constraints. The following postulates are satisfied for any \( c, d \in U \) and any \( X, Y, Z \in \text{Var} \).

1. the structure \( (U, \sqsubseteq) \) forms a domain.
2. \( \exists x \) is a continuous function \( \exists x : \mathcal{U} \to \mathcal{U} \Gamma \)
3. \( \exists x (\top) = \top \Gamma \)
4. \( \exists x (c) \subseteq c \Gamma \)
5. \( \exists x (c \cup \exists x (d)) = \exists x (c) \cup \exists x (d) \Gamma \)
6. \( \exists x (\exists y (c)) = \exists y (\exists x (c)) \Gamma \)
7. \( (X = X) = \bot \Gamma \)
8. \( (X = Y) = \exists z (X = Z \cup Z = Y) \Gamma \) for \( Z \) distinct from \( X \) and \( Y \Gamma \)
9. \( c \subseteq (X = Y) \cup \exists x (X = Y \cup c) \).

Items 3-9 are borrowed from cylindric algebra. However, the structure is not necessarily a cylindric algebra since we do not require an inverse operation.

4 The Language

We assume that the following is given. A set of variables \( X, Y \ldots \in \text{Var} \) a constraint system \( \langle \mathcal{U}, \subseteq, X = Y, \exists x \rangle_{X, Y \in \text{Var}} \) and a set of procedure symbols \( P, \ldots \in \mathcal{N} \).

4.1 Syntax

The set of agents \( \Gamma \) is as follows.

\[
A ::= c | \bigwedge_{j \in I} A^j | (c_1 \Rightarrow A_1 [ \ldots [ c_n \Rightarrow A_n ] \ldots ] | \exists x A | P(X)
\]

A tell constraint written \( c \Gamma \) is assumed to be a member of \( \mathcal{K} (\mathcal{U}) \).

The conjunction

\[
\bigwedge_{j \in I} A^j
\]

of agents where \( I \) is assumed to be a countable set of integers represents a parallel composition of the agents \( A^j \).

We will use \( A_1 \land A_2 \) as a shorthand for \( \bigwedge_{j \in \{1, 2\}} A^j \).

Agents of the form \( \exists x A \) existentially quantified agents are used to describe agents with local data \( c \). If the local data is \( \bot \) we will often omit the local data and write \( \exists x A \).

Existentially quantified agents occurring in programs or in the beginning of a computation are required to be of the form \( \exists x A \Gamma \) i.e., \( \Gamma \) to have no information stored locally.

An agent

\[
(c_1 \Rightarrow A_1 [ \ldots [ c_n \Rightarrow A_n ] \ldots ])
\]

represents a selection. If one of the conditions \( c_k \) becomes true, the corresponding alternative \( A_k \) may be executed.

An agent of the form \( P(Y) \) represents a procedure call.

A program \( \Pi \) is a set of definitions of the form \( P(X) :: \Gamma \) where each procedure symbol \( P \) occurs in the left-hand side of exactly one definition in the program.
4.2 Oracles

An oracle is a finite or infinite string over $\omega$. Let $\text{ORACLE}$ be the set of oracles. For oracles $s$ and $s'$ let $s \leq s'$ denote that $s$ is a prefix of $s'$.

We will use the notation $k.s$ for an oracle where the first element is $k$ and the following elements are those given by $s$.

When giving the semantics of a conjunction $\Gamma$ we need a way to distribute the oracle to the agents in the conjunction. Since infinite conjunctions are allowed we must be able to distribute an oracle into an infinite set of oracles.

We begin by defining the functions $\text{EVEN}, \text{ODD} : \text{ORACLE} \rightarrow \text{ORACLE}$ according to

$$\text{EVEN}(s) = k_0 k_2 k_4 \ldots$$
$$\text{ODD}(s) = k_1 k_3 k_5 \ldots$$

for $s = k_0 k_1 k_2 \ldots$.

Define functions $\pi_n : \text{ORACLE} \rightarrow \text{ORACLE}$ over oracles $\Gamma$ for $n \in \omega \Gamma$ according to the rules

$$\pi_0 s = \text{EVEN}(s)$$
$$\pi_{n+1} s = \pi_n(\text{ODD}(s))$$

It is easy to see that if we have a family of oracles $\{s_n\}_{n \in \omega}$ there is an oracle $s$ such that $\pi_n s = s_n \Gamma$ for $n \in \omega$.

4.3 Computation Rules

A configuration is a triple $A(s) : c$ consisting of an agent $A \Gamma$ an oracle $s \Gamma$ and a finite constraint $c$ (the store). The oracle will sometimes be omitted when it is either given by the context or not relevant. We define a binary relation $\rightarrow$ over configurations according to Figure 1.

The tell constraint $s$ simply adds new information (itself) to the store.

A conjunction $\bigwedge_{j \in I} A^j$ performs a computation step by performing a computation step by one of its components.

If one of the ask constraints in a selection is satisfied by the current store $\Gamma$ the selection can be reduced to the corresponding agent. Note that it is required that the choice conforms with the oracle.

A configuration with an existentially quantified agent $\exists X A : d$ is executed one step by doing the following. Apply the function $\exists X$ to the present store (given by $d$) hiding any information related to the variable $X$. Combine the result $\exists X(d)$ with the local data (given by $c$) to obtain a local store. A computation step is performed in the local store which gives a new local store ($c'$ say). To transmit any results to the global store $\Gamma$ the function $\exists X$ is again applied to hide any information related to the variable $X$. The constraint thus obtained is combined with the previous global store $d$.

A call $P(X)$ is executed one step by replacing the call by the body of the definition of the procedure $\Gamma$ enclosed in two nested existential quantifications which represent a substitution of the formal parameter for the argument of the call.

If the definition if of the form $P(Y) : A \Gamma$ the call $P(X)$ will be replaced by $\exists \Delta (\Delta = X \land \exists Y (\Delta = Y \land A)) \Gamma$ which corresponds to first creating a local scope for the variable $\Delta \Gamma$ binding $\Delta$ to the value of $XT$ then creating a local scope for the
1. \( c(s) : d \rightarrow c(s) : c \cup d \)

2. \[
\frac{\forall j \in I A^{k} s : c \rightarrow B^{k}(\pi_{k} s') : d, \ k \in I}{\bigwedge_{j \in I} A^{I} s : c \rightarrow \bigwedge_{j \in I} B^{I}(s') : d},
\]
where \( B^{j} = A^{j} \) and \( (\pi_{j} s') = (\pi_{j} s) \Gamma \) for \( j \in I \setminus \{k\} \).

3. \[
\frac{c_{k} \subseteq c}{(c_{1} \Rightarrow A_{1} \bigg| \ldots \bigg| c_{n} \Rightarrow A_{n})(k,s) : c \rightarrow A_{k}(s) : c}
\]

4. \[
\frac{c \subseteq \exists \times (d) \rightarrow A'(s') : c'}{\exists \times A(s) : d \rightarrow \exists \times A'(s') : d \cup \exists \times (c')}
\]

5. \[
P(X)(s) : c \rightarrow A\{X/Y\}(s) : c,
\]
where \( \Pi \) contains \( P(Y) : A \)
and \( A\{X/Y\} \equiv \exists \Delta(\Delta = X \cup \exists Y(\Delta = Y \wedge A)) \).

Figure 1: Computation rules

variable \( Y \) and then binding \( Y \) to the value of \( \Delta \). Finally the agent \( A \) is executed inside the local scope of \( Y \).

The doubly nested quantifiers are necessary to handle the case where the variables \( X \) and \( Y \) are the same.

### 4.4 Configurations and Computation Rules

First we give the set of computations for a given program.

**Definition 4.1** Assuming a program \( \Pi \) a *computation* is an infinite sequence of configurations \( (A_{i}(s_{i}) : c_{i})_{i \in \omega} \) such that for all \( i \geq 0 \) we have either \( A_{i}(s_{i}) : c_{i} \rightarrow A_{i+1}(s_{i+1}) : c_{i+1} \) (a *computation step*) or \( A_{i}(s_{i}) = A_{i+1}(s_{i+1}) \Gamma \) and \( c_{i} \subseteq c_{i+1} \) (an *input step*).

An input step from \( A(s) : c \) to \( A(s) : c' \Gamma \) such that \( c = c' \) is an *empty input step*.

For an agent \( A \) and an oracle \( s \) an \( A\{s\} \)-computation is a computation \( (A_{i}(s_{i}) : c_{i})_{i \in \omega} \) such that \( A_{0}(s_{0}) = A(s) \).

In the following text we will leave out references to the program \( \Pi \) when we can do so without causing ambiguities.

Note that even though all computations are infinite we can represent a finite computation by a computation with an infinite tail of empty input steps.

### 4.5 Fairness

The structured operational semantics does not in itself define fairness. It is necessary to use some device to restrict the set of computations thus avoiding for example situations where one agent in a conjunction is able to perform a computation step but is never allowed to do so.
Intuitively, a computation is fair if every agent that occurs in it and is able to perform some computation step will eventually perform some computation step. However, this intuitive notion is difficult to formalise directly. What does it mean that an agent is able to perform a computation step? Computation steps are performed on configurations, not on agents. Also, this requirement should not apply to alternatives in a selection since an agent occurring in an alternative should not be executed until (and if) that alternative is selected. Third, what happens if one has a computation where an agent A occurs in many positions in every configuration in the computation? The intuitive fairness requirement does not differentiate between different occurrences of the same agent, so a computation might incorrectly be considered fair if it performed computation steps on some occurrences of the agent A and ignored other occurrences of A.

A computation can often be considered to contain other computations. For example, to perform a computation step with a process \( A \leq B : c \) it is necessary to perform computation steps with either of the processes \( A : c \) and \( B : c \). The view of a computation as a composition of computations leads us to the following definitions.

**Definition 4.2** The relation immediate inner computation of is the weakest relation over \( \omega \)-sequences of configurations that satisfies the following:

1. \( (A^k_i \sigma_i s_i) : c_i) \in \omega \) is an immediate inner computation of \( (\bigwedge_{i \in I} A^k_i(s_i) : c_i)_{i \in \omega} \Gamma \) for \( k \in I \).

2. \( (A_i(s_i) : c_i \sqcup \exists X(d_i))_{i \in \omega} \) is an immediate inner computation of the computation \( (\exists X A_i(s_i) : d_i)_{i \in \omega} \).

The relation ‘inner computation of’ is defined to be the transitive closure over the relation ‘immediate inner computation of’.

**Proposition 4.3** If \( (A_i(s_i) : c_i)_{i \in \omega} \) is a computation and \( (B_i(s'_i) : d_i)_{i \in \omega} \) is an inner computation of \( (A_i(s_i) : c_i)_{i \in \omega} \) then \( (B_i(s'_i) : d_i)_{i \in \omega} \) is also a computation.

**Definition 4.4** A computation \( (A_i(s_i) : c_i)_{i \in \omega} \) is top-level fair when the following holds:

1. If \( A_0 = p(X) \) there is an \( i \geq 0 \) such that \( A_i \neq A_0 \).
2. If \( A_0 = c \) there is an \( i \geq 0 \) such that \( c_i \sqcup c \).
3. If \( A_0 = (d_1 \Rightarrow B_1 \sqcup \ldots \sqcup d_n \Rightarrow B_n) \Gamma \) and \( s = k.s' \) where \( k \geq 0 \) then there is an \( i \geq 0 \) such that \( A_i = B_k \).
4. If \( A_0 = (d_1 \Rightarrow B_1 \sqcup \ldots \sqcup d_n \Rightarrow B_n) \Gamma \) and \( s = 0.s' \Gamma \) then \( d_j \not\in c_i \) for all \( j \leq n \) and \( i \geq 0 \).

A computation is initially fair if all its inner computations are top-level fair. A computation is fair if all its proper suffixes are initially fair.

**Proposition 4.5** If one suffix of a computation is top-level fair then the computation is top-level fair.
Proof. Consider a computation \((A_i : c_i)_{i \in \omega}\) such that \((A_i : c_i)_{i \geq k}\) is top-level fair. If \(A_0 = p(Y)\) we have two possible cases. If \(A_k = p(Y)\) there must be a \(j > k\) such that \(A_j \neq A_k\) since the suffix is top-level fair. Otherwise \(A_k \neq A_0\) so the computation is top-level fair also in this case.

If \(A_0 = c\) then by the reduction rules \(A_k = c\) and there is a \(j \geq k\) such that \(c_j \sqsubseteq c\).

The case when \(A_0 = (d_1 \Rightarrow B_1 \square \ldots \square d_n \Rightarrow B_n)\) is similar. □

4.5.1 Informal Justification Recall that an intuitive notion of fairness was proposed which said that a computation is fair if every agent that occurs in the computation and is able to perform a computation step will eventually perform some computation step.

We will attempt to justify the formal definition of fairness by giving an argument to why it conforms with the intuitive notion. Note that the formal definition of fairness involves oracles but that the intuitive notion is independent of oracles. We argue that if a computation is fair in the formal sense it is also fair in the intuitive sense and if a computation is fair in the intuitive sense it is also fair in the formal sense given an appropriate choice of oracle.

Suppose that we have a computation \(\Gamma\) which is fair in the formal sense. Consider an agent \(A\) which occurs somewhere in the computation. Consider the suffix \(\Gamma'\) of the computation \(\Gamma\) in which the agent \(A\) occurs in the first configuration. By the definition of fairness the computation \(\Gamma'\) is initially fair. This means that every inner computation of \(\Gamma'\) must be top-level fair in particular that any inner computation which begins with the configuration in which \(A\) is an agent is top-level fair. If \(A\) is a tell constraint \(c\) top-level fairness means that the corresponding store must eventually entail \(c\). This does not necessarily mean that \(A\) will perform a computation step but the end result will be the same. If \(A\) is a call or a selection in which one of the conditions are entailed top-level fairness means that \(A\) will eventually perform a computation step. If \(A\) is a conjunction or an existentially quantified agent know from the computation rules that \(A\) contains some agent \(A'\) which is either a call a tell constraint or a selection with an enabled condition. Again top-level fairness means that \(A'\) must eventually perform some computation step. Since the computation rules imply that a conjunction or an existentially quantified agent performs computation steps exactly when some internal agent performs a computation step \(\Gamma\) it follows that \(A\) is forced by the formal definition of fairness to perform a computation step.

In a similar fashion assuming that a computation \(\Gamma\) is fair in the intuitive sense we argue that it should also be fair in the formal sense. Recall that a computation is fair in the formal sense only if all its suffixes are initially fair. Now considering the computation \(\Gamma\) clearly any suffix \(\Gamma'\) of \(\Gamma\) is also fair in the intuitive sense. We now want to show that each inner computation of \(\Gamma'\) is top-level fair. Suppose \(A\) occurs in the first configuration of \(\Gamma'\) (we assume that there is only one occurrence of \(A\)). If \(A\) is a tell constraint \(c\) it follows from the intuitive notion of fairness \(A\) should eventually perform some computation step and thus there should be some future store which contains the constraint \(c\). If \(A\) is a call the intuitive notion of fairness gives that \(A\) should perform a computation step \(\text{i.e.}\ \Gamma'\) be replaced by the body of the corresponding definition. If \(A\) is a selection in which one of the conditions is entailed \(\Gamma'\) it follows that \(A\) is able to perform a computation step and thus will eventually do so. If we assume that the oracle begins with \(\Delta\) where the
$k$th alternative is the one which will be selected if we find that the computation is top-level fair. Suppose $A$ is a selection which will never perform a computation step. We conclude that none of the conditions is ever entailed by the store. In particular, that none of the conditions is entailed by the first store. We assume the oracle to begin with $0$ and find that the computation beginning with $A$ is top-level fair.

It follows that the inner computation beginning with $A$ is top-level fair. It follows that every inner computation of $\Gamma'$ is top-level fair. Thus $\Gamma'$ is initially fair. We draw the conclusion that every $\Gamma$ is initially fair and it follows that $\Gamma$ is fair.

### 4.5.2 Properties of fairness

We give a few properties of fairness relating fairness of a computation with fairness of its suffixes and inner computations. The properties are should be intuitively clear.

**Proposition 4.6** If one suffix of a computation is initially fair, then the computation is initially fair.

*Proof.* Suppose we have a computation $x$ where the $k$th suffix is initially fair. Consider an inner computation $y$ of $x$. The $k$th suffix of $y$ is an inner computation of the $k$th suffix of $x$ and thus top-level fair. So by proposition 4.5 every inner computation of $x$ is top-level fair and therefore $x$ is initially fair.

**Lemma 4.7** If one suffix of a computation is initially fair, then the computation is fair.

*Proof.* Let $x$ be a computation with one suffix $y$ which is fair. Let $z$ be a suffix of $x$. If $z$ is also a suffix of $y$, the computation $z$ is initially fair since $y$ is fair. If, on the other hand, $y$ is a suffix of $z$, $z$ must be initially fair since it has a suffix which is initially fair. Each suffix of $x$ is initially fair so $x$ must be fair.

**Lemma 4.8** All inner computations of a fair computation are fair.

*Proof.* Suppose that $x$ is a fair computation and that $y$ is an inner computation of $x$. Let $k$ be fixed. The $k$th suffix of $y$ is an inner computation of the $k$th suffix of $x$, which is initially fair. So the $k$th suffix of $y$ and all its all its inner computations are top-level fair which implies that the $k$th suffix of $y$ is initially fair. So each suffix of $y$ is initially fair and thus is $y$ fair.

### 5 Result and Trace Semantics

Given the operational model of concurrent constraint programming presented in the earlier chapters, one can immediately give an operational semantics. In this chapter we give three versions; first the result semantics which considers only the relation between the initial and final constraint stores in a computation. Obviously, the result semantics provides a minimal amount of information that should also be provided by any reasonable semantics. The result semantics is of course not compositional since it does not capture interaction between agents.

The second semantics is the trace semantics where a process is represented by a set of traces. Each trace is an infinite sequence of environments together with information on which steps in the computation are computation steps and which
are input steps. Since the trace semantics records interaction between processes one would expect the trace semantics to be compositional and this is indeed the case [16].

The third semantics the *abstract semantics* is based on the trace semantics. Note that the trace semantics is not fully abstract since for example it records the number of steps that is required for an agent to produce a result. The abstract semantics is defined through a simple closure operation over sets of traces and has been shown to be fully abstract and compositional [16].

### 5.1 Results

The result semantics is given by a function \( \mathcal{R}_H : \text{AGENT} \times \text{ORACLE} \rightarrow K(\mathcal{U}) \rightarrow \wp(\mathcal{U}) \) which gives the set of all possible results that can be computed given a program \( \Pi \) an agent \( A \) and an initial environment \( c \).

\[
\mathcal{R}_H [A(s)]c = \bigcup_{i \in \omega} c_i \mid (A_i(s_i) : c_i)_{i \in \omega} \text{ is a fair non-interactive \( A(s) \)-computation with } c_0 = c \}
\]

Note that for some combinations of \( A \) and \( s \) the result semantics may be an empty set if the oracle requires a choice that is not possible to make.

It can be seen that for an infinite computation we define the final constraint store as the limit of the intermediate constraint stores that occur during a computation. We find this very reasonable for a constraint programming language since an arbitrary finite approximation of the "final" constraint store can be obtained by waiting long enough for the computation to proceed. This property does not hold for shared-variable programs in general where the information in the store does not have to be monotonously increasing.

To reflect the result semantics of an agent which does not depend on an oracle we give the following definition.

\[
\mathcal{R}_H [A]c = \bigcup_{s \in \text{ORACLE}} \mathcal{R}_H [A(s)]
\]

### 5.2 Traces

**Definition 5.1** A trace \( t \) is a pair \( t = (v(t), r(t)) \) where \( v(t) \) is an \( \omega \)-chain in \( K(\mathcal{U}) \) and \( r(t) \subseteq \omega \). The set of traces is denoted \( \text{TRACE} \).

The trace of a computation \( (A_i(s_i) : c_i)_{i \in \omega} \) is a trace \( t = ((c_i)_{i \in \omega}, r) \) where the step from \( A_i(s_i) : c_i \) to \( A_{i+1}(s_{i+1}) : c_{i+1} \) is a computation step when \( i \in r \) and an input step when \( i \not\in r \). For \( t \in \text{TRACE} \) \( v(t) \) will sometimes be referred to as the store sequence of the trace. We will sometimes use the notation \( v(t)_i \) to refer to the \( i \)th element of the store sequence of \( t \).

The trace semantics of an agent \( A \) together with an oracle \( s \) assuming a program \( \Pi \) is defined as follows.

**Definition 5.2**

\[
\mathcal{O}_H [A(s)] = \{ t \in \text{TRACE} \mid t \text{ is the trace of a fair } A(s)\text{-computation.} \}
\]

\[
\mathcal{O}_H [A] = \bigcup_{s \in \text{ORACLE}} \mathcal{O}_H [A(s)]
\]
5.3 Compositionality

The trace semantics allows a compositional definition as expressed in the following propositions.

**Proposition 5.3** $t \in \mathcal{O}_\Pi \llbracket \bigwedge_{j \in I} A_j \rrbracket$ iff there are $t_j \in \mathcal{O}_\Pi \llbracket A_j \rrbracket$ for $j \in I$ such that $v(t_j) = v(t)$ for $j \in I$ and $r(t) = \bigcup_{j \in I} r(t_j)$ with $r(t_i) \cap r(t_j) = \emptyset$ for $i, j \in I$ such that $i \neq j$.

*Proof.* (⇒) We know that there is a fair computation $\left( \bigwedge_{j \in I} A^j_i : c_i \right) \in \omega$ that connects $t$ to $A$. By the definition of the reduction rules, we have a family of computations $\left( \left( A^j_i : c_i \right)_{i \in \omega} \right)$ such that for each $i \in r(t)$ there is a $k_i \in I$ such that $A^k_i : c_i \rightarrow A^{k+1}_i : c_{i+1}$ is a reduction step. Let $j \in I \setminus \{ k_i \}$ then there is an input step from $A^j_i : c_i$ to $A^{j+1}_i : c_{i+1}$. For $i \in \omega \setminus r(t)$ it is easy to see that the step from $A^j_i : c_i$ to $A^{j+1}_i : c_{i+1}$ must be an input step for all $j \in I$.

That for all $j \in I$ each computation $\left( A^j_i : c_i \right)_{i \in \omega}$ is fair follows from Lemma 4.8 which says that all inner computations of a fair computation are fair. For each $j \in I$ let the trace $t_j$ be such that $v(t_j) = v(t) \Gamma$ and $r(t_j) = \{ j \in r(t) \mid k_i = j \}$. It is easy to check that the family of traces $\left( t_j \right)_{j \in I}$ satisfies the right-hand side of the proposition.

(⇐) We know that for each $j \in I$ there is a fair computation $\left( A^j_i : c_i \right)_{i \in \omega}$ that connects $t_j$ to $A_j$. For each $i \in r(t)$ there is a $k_i \in I$ such that $i \in r(t(k_i))$ but $i \notin r(t_j) \Gamma$ for $j \in I \setminus \{ k_i \}$. By the computation rules it follows that

$$\bigwedge_{j \in I} A^j_i : c_i \rightarrow \bigwedge_{j \in I} A^{j+1}_i : c_{i+1},$$

for all $i \in r(t)$. In the case that $i \in \omega \setminus r(t) \Gamma$ it follows that $i \in \omega \setminus r(t_j) \Gamma$ for all $j \in I$ and thus all computations $\left( A^j_i : c_i \right)_{i \in \omega}$ perform input steps at position $i \Gamma$ which implies that $\bigwedge_{j \in I} A^j = \bigwedge_{j \in I} A^j \Gamma$ from which follows that we can construct a computation $\left( \bigwedge_{j \in I} A^j_i : c_i \right)_{i \in \omega}$. Fairness follows from the fact that all immediate inner computations of the constructed computation are fair.

**Proposition 5.4** Suppose we have an agent $\exists X A$. For any trace $t \Gamma$ we have $t \in \mathcal{O}_\Pi \llbracket \exists X A \rrbracket$ iff there is a trace $u \in \mathcal{O}_\Pi \llbracket A \rrbracket$ such that with $v(t) = (d_i)_{i \in \omega}$ and $v(u) = (e_i)_{i \in \omega} \Gamma$ we have

1. $r(t) = r(u)$
2. $e_0 = \exists X (d_0) \Gamma$
3. for $i \in r(t) \Gamma d_{i+1} = d_i \cup \exists X (e_{i+1}) \Gamma$ and
4. for $i \in \omega \setminus r(t) \Gamma e_{i+1} = e_i \cup \exists X (d_{i+1})$.

*Proof.* (⇒) Suppose $t \in \mathcal{O}_\Pi \llbracket \exists X A \rrbracket$. There is a computation $\left( \exists X A_i : d_i \right)_{i \in \omega}$ that connects the trace $t$ to the agent $A$. The computation has an inner computation $\left( A_i : c_i \cup \exists X (d_i) \right)_{i \in \omega}$ that we know by Propositions 4.3 and 4.8 to be a fair computation. It remains to prove that with $e_i = c_i \cup \exists X (d_i)$ conditions 1-4 are satisfied.
When i / ε r(t)Γ it follows that ΞX dX ⌢ i+1 : dX ⌢ i+1 , and by the computation rules that A i : cX(dX) i+1 → cX(dX) i and dX ⌢ i+1 = dX ⌢ i+1 . By the properties of the constraint system we have dX ⌢ i+1 = dX ⌢ i+1 ∪ ΞX(dX) i+1 = dX ⌢ i+1 ∪ ΞX(cX(dX) i+1 ). Condition 3 follows immediately.

If i ε ω \ r(t) the corresponding step in the computation (ΞX A i : dX) i ε ω is an input step. This implies that cX i = cX i+1 . So eX i+1 = eX i+1 ∪ ΞX(dX) i+1 = cX i ∪ ΞX(dX) i+1 = eX i ∪ ΞX(dX) i+1 .

(=) Assume that the right-hand side of the proposition holds. There is a computation (A i : eX) i ε ω that connects the trace u to the agent A. For all i ε ω \ u let cX i = ∪{eX j+1 | j < i, j ε r(t)} (this should agree with the idea that cX i, which is the local data of the agent, only changes when the agent performs computation steps).

We want to show that (ΞX A i : dX) i ε ω is a fair computation that connects the trace r(t) to the agent A. Note that for all i ε ω \ r(t) it follows from our assumptions that ΞX(dX) i = ΞX(eX i ) and cX i ∪ ΞX(dX) i = eX i . If i ε r(t) = r(u)Γ we know that cX i+1 = eX i+1 and A i : eX i → A i+1 : eX i+1 . By the computation rules and the equalities above Γ ΞX A i : dX i → ΞX A i+1 : dX i+1 .

If i ε ω \ r(t) we have A i+1 = A i and cX i+1 = cX i so the i th step of (ΞX A i : dX) i ε ω is an input step.

To establish fairness of the computation (ΞX A i : dX) i ε ω it suffices to observe that its only immediate inner computation is fair.

**Proposition 5.5** t ∈ ⋄[c1 ⇒ A1 [.. . .. c n ⇒ A n ]] if one of the following holds.

1. c j ⊆ v(t) k for some j ≤ n and k ≥ 0 and there is a u ∈ ⋄[A j ] such that for all i ≥ 0 v(u) i = v(t) i+k+1 Γ v(t) k = v(t) k+1 Γ and r(t) = {i + k + 1 | i ∈ r(u)} ∪ {k}.

2. There is no j ≤ n and k ≥ 0 such that c j ⊆ v(t) k Γ and r(t) = ∅.

The proof follows directly from the operational definition.

### 5.4 Closure operators

In the definition of the abstract semantics in the following section Γ and of the fixpoint semantics which is the main theme of this paper the concept of closure operators plays an important role. In this section we introduce the concept and show some of its properties.

The closure operators capture important aspects of computations in constraint programming languages. One early definition of the concept of closure operators is by Moore [13Γ pages 53–80]. Jagadeesan and Panangaden and Pingali [5] showed how a concurrent process operating over a domain that allows ‘logic variables’ Γ place holders for values that are to be defined later could be viewed as a closure operator. This idea was explored in a concurrent constraint programming setting by Saraswat, Rinard and Panangaden [19].

The following section gives the definition of closure operators and some of their properties.
Definition 5.6 For a domain $(D, \sqsubseteq)$ a closure operator over $D$ is a function $f$ over $D$ with the property that $f(x) \supseteq x$ and $f(f(x)) = f(x)$ for any $x$ in $D$. A continuous closure operator is a closure operator which is also continuous.

The set of fixpoints of a closure operator $f$ over a domain $D$ is the set $f(D)$. Suppose that $S$ is the set of fixpoints of a closure operator $f$ that is $S = f(D)$ where $D$ is the domain of $f$. For any subset $T$ of $S \cap T \in S$. This is easy to see if we observe that $f(\cap T) \subseteq \cap \{f(x) \mid x \in T\}$ since $f$ is monotone. $\cap T \subseteq f(\cap T)$ since $f$ is a closure operator and $\cap \{f(x) \mid x \in T\} = \cap T$ since all members of $T$ are fixpoints of $f$. It follows that the set of fixpoints of a closure operator is closed under greatest lower bounds of directed sets.

On the other hand, if $S \subseteq D$ is such that $S$ is closed under arbitrary greatest lower bounds. We can define a function $f_S$ according to the rule $f_S(x) = \cap \{x \cap S\}$ i.e., let $f_S(x)$ be the least element of $S$ greater than $x$. It is easy to see that the function $f_S$ is well-defined and a closure operator.

Thus there is a one-to-one correspondence between closure operators and sets closed under $\cap$. In the subsequent text we will take advantage of this property and sometimes see closure operators as functions and sometimes as sets so to say that $x$ is a fixpoint of the closure operator $f$ we can write $x = f(x)$ or $x \in f$.

Next we will show that the closure operators over a domain form a complete lattice. If we consider the functions over a domain to be ordered point-wise i.e., $f \sqsubseteq g$ if and only if $f \sqsubseteq g$ for all $x \in \Gamma$ we have $f \sqsubseteq g$ if and only if $f \sqsubseteq g$. If $\{f_i\}_{i \in I}$ is a family of closure operators it is easy to see that $\cap_{i \in I} f_i$ is also a closure operator: this is obviously the least upper bound of the family of closure operators. The top element of the lattice of closure operators over a domain $D$ is the function that maps every element of the domain to $\top$ and the bottom element is the identity function.

For an element $x \in D$ and a closure operator $f$ over $D \\Gamma$ we define $(x \to f)$ as the closure operator given by

$$(x \to f)(y) = \begin{cases} f(y), & \text{if } y \supseteq x \\ y, & \text{otherwise.} \end{cases}$$

Since a closure operator is characterised by its set of fixpoints $\\Gamma$ the following definition will also suffice.

$$(x \to f) = f \cup \{y \mid x \not\sqsubseteq y\}$$

Similarly for elements $x \in D$ and $y \in D$ the closure operator $(x \to y)$ is defined as follows

$$(x \to y) = (x \to y \uparrow),$$

where $y \uparrow$ is the closure operator whose set of fixpoints is $\{z \in D \mid y \sqsubseteq z\}$. Note that when $x$ is finite the closure operators $(x \to f)$ and $(x \to y)$ are continuous. For $f$ continuous and arbitrary $y$.

Unless stated otherwise we will assume the closure operators occurring in this paper to be continuous.

We can generalise the existential quantifier to be defined on closure operators. For a closure operator $f$ let $E_X(f)$ be function defined as follows.

$$E_X(f) = (\exists_X \circ f \circ \exists_X) \cup \text{id}$$

Clearly $E_X(c)$ is a closure operator.
**Proposition 5.7** For a closure operator $\Gamma$ the closure operator $E_X(f)$ has the set of fixpoints given by the following equation.

$$E_X(f) = \{ c \mid \text{There is a constraint } d \in f \text{ such that } \exists_X(c) = \exists_X(d) \}$$

*Proof.* Note that for any constraint $c\Gamma$ we have $E_X(f)c \subseteq f(c)$. From this follows that any fixpoint of $f$ must also be a fixpoint of $E_X(f)$. Let $g = E_X(f)$.

(⊇) Suppose we have a constraint $d \in f$. It follows that $d = g(d)$. Let $c$ be a constraint such that $\exists_X(c) = \exists_X(d)$. Applying $g$ gives $g(c) = \exists_X(f(\exists_X(c))) \sqcup c = \exists_X(f(\exists_X(d))) \sqcup c \subseteq \exists_X(f(d)) \sqcup c = \exists_X(d) \sqcup c = \exists_X(c) \sqcup c = c$.

(⊆) Suppose that $c$ is a fixpoint of $g$. Let $d = f(\exists_X(c))$. By idempotence of $\Gamma$ the constraint $d$ is a fixpoint of $f$ and since $c = g(c)\Gamma$ we must have $c \sqcup \exists_X(f(\exists_X(c))) = \exists_X(d)$. So $\exists_X(c) \sqcup \exists_X(d)\Gamma$ and since $f(\exists_X(c)) \sqsubseteq \exists_X(c)\Gamma$ which implies $\exists_X(c) \subseteq \exists_X(d)$ we have $\exists_X(c) = \exists_X(d)$. □

We generalise the $E_X$ operator to be defined over arbitrary sets $S$ by

$$E_X(S) = \{ c \mid \text{There is a constraint } d \in f \text{ such that } \exists_X(c) = \exists_X(d) \}.$$  

### 5.5  A fully abstract semantics

The idea is that we look at two aspects of a trace: its functionality and its limit. The limit of a trace $t$ is simply the limit of the sequence of store of the trace $t$ that is $\Gamma$:

$$\lim(t) = \bigcup_{i \in \omega} v(t)_i.$$  

The *functionality* of a trace $t\Gamma$ denoted $\text{fn}(t)\Gamma$ is the closure operator given by the following equation.

$$\text{fn}(t) = \bigcap_{i \in r(t)} (v(t)_i \rightarrow v(t)_{i+1})$$  

Say that a trace $t$ is a *subtrace* of a trace $t'$ if $\lim t = \lim t'$ and $\text{fn} t \subseteq \text{fn} t'$. We can now define the abstract semantics.

**Definition 5.8** For an agent $A\Gamma$ and a program $\Pi\Gamma$ let

$$A_{\Pi}[A] = \{ t \mid t \text{ is a subtrace of } t'\Gamma \text{ for some } t' \in \mathcal{O}_{\Pi}[A] \}.$$  

□

The abstract semantics contains sufficient information to allow the result semantics to be obtained from the abstract semantics. The abstract semantics is also compositional $\Gamma$ and has been shown to satisfy the equations of Figure 2.

### 5.6  An example

Consider the semantics of the agent the agent

$$A = (X = 1 \Rightarrow Z = 3 \parallel Y = 2 \Rightarrow W = 5).$$
\[
\mathcal{A}_t[c] = \{ t \mid \text{fn}(t) \supseteq (\bot \rightarrow c) \}
\]
\[
\mathcal{A}_t[\bigwedge_{j \in I} A_j] = \{ t \mid t_j \in \mathcal{A}_t[A_j] \text{ and } \text{lim}(t_j) = \text{lim}(t), \text{ for } j \in I, \text{fn}(t) \supseteq \bigcap_{j \in I} \text{fn}(t_j) \}
\]
\[
\mathcal{A}_t[\exists X . A] = \{ t \mid t' \in \mathcal{A}_t[A], \exists X (\text{lim}(t)) = \exists X (\text{lim}(t')), \text{lim}(u) = (\text{fn}(u) \circ \exists X) \text{lim}(u), \text{ and } \text{fn}(t) \supseteq E_X (\text{fn}(t')) \}
\]
\[
\mathcal{A}_t[\bigcap_{k \leq n} c_k \Rightarrow A_k] = \{ t \mid \text{fn}(t) = \text{id} \text{ and } \text{lim}(t) \sqsubseteq c_k, \text{ for } k \leq n \} \cup \{ t \mid k \leq n, t' \in \mathcal{A}_t[A_k], \text{lim}(t) = \text{lim}(t') \sqsubseteq c_k, \text{ and } \text{fn}(t) = (c_k \rightarrow \text{fn}(t')) \}
\]
\[
\mathcal{A}_t[P(X)] = \mathcal{A}_t[A[X/Y]], \text{ where the definition of } P \text{ is } P(Y) :: A
\]

Figure 2: The abstract semantics as a set of equations

The result semantics is

\[
\mathcal{R}_t[A(s)]c = \{ c \}, \quad \text{for oracles } s = 0 \ldots \text{ and constraints } c \not\supseteq (X = 1), (Y = 2)
\]
\[
\mathcal{R}_t[A(s)]c = \emptyset, \quad \text{for oracles } s = 0 \ldots \text{ and constraints } c \supseteq (X = 1) \sqcup (Y = 2)
\]
\[
\mathcal{R}_t[A(s)]c = \{ c \sqcup (Z = 3) \}, \quad \text{for oracles } s = 1 \ldots \text{ and constraints } c \supseteq (X = 1)
\]
\[
\mathcal{R}_t[A(s)]c = \{ c \sqcup (W = 5) \}, \quad \text{for oracles } s = 2 \ldots \text{ and constraints } c \supseteq (Y = 2)
\]
\[
\mathcal{R}_t[A(s)]c = \emptyset, \quad \text{in all other cases}
\]

We see that the result semantics is either an empty set or a singleton set. This suggests that it should be possible to consider an agent with a given oracle to be deterministic.

The trace semantics of \( \mathcal{O}_t[A(s)] \) is even for this very simple agent hard to describe in a concise manner. We will just give an example of a typical trace. Let \( s = 111 \ldots \). We have \( t \in \mathcal{O}_t[A(s)] \) where

\[
\begin{align*}
\text{r}(t) &= (\bot, \bot, X = 1, X = 1, X = 1, (X = 1) \sqcup (Z = 3), \ldots) \\
\text{r}(t) &= \{ 3, 4 \}
\end{align*}
\]

The trace only has two computation steps. In the first step nothing happens (an empty input step). In the following step the constraint \( X = 1 \) is input \( \Gamma \text{id} \) added to the store `from the outside’. In the next step nothing happens. In step number \( 3 \) the agent performs a computation step \( \Gamma \) without altering the store. (This step is of course when the agent selects the first alternative.) In step number \( 4 \) the agent adds the constraint \( Z = 3 \) to the store. After this nothing more happens.

Other traces for \( A(s) \) might involve input of constraints that are not relevant to the computation. In general the set of traces for an agent is uncountable.

The abstract semantics of the agent \( A = (X = 1 \Rightarrow Z = 3 \, \sqcup \, Y = 2 \Rightarrow W = 5) \) is
summarised in the following four rules.

\[ A_H[A(s)] = \{ t \mid \text{fin } t = \text{id}, \lim t \notin (X = 1)^u \cup (Y = 2)^u \}, \]
for oracles \( s = 0 \ldots \)

\[ A_H[A(s)] = \{ t \mid \text{fin } t \subseteq \lambda c.(X = 1 \rightarrow Z = 3), \]
\[ \lim t \in (X = 1)^u \cap (Z = 3)^u \}, \]
for oracles \( s = 1 \ldots \)

\[ A_H[A(s)] = \{ t \mid \text{fin } t \subseteq \lambda c.(Y = 2 \rightarrow W = 5), \]
\[ \lim t \in (Y = 2)^u \cap (W = 5)^u \}, \]
for oracles \( s = 2 \ldots \)

\[ A_H[A(s)] = \emptyset, \]
for oracles \( s = k \ldots \Gamma \) with \( k \geq 3 \)

Note how the traces can be easily classified in three groups with respect to functionality and limit.

6 Confluence

The behaviour of an agent with a given oracle is essentially deterministic. For each selection at most one alternative may be chosen. This suggests the existence of some confluence property. In this chapter we consider both a basic confluence property regarding finite computation sequences and a more general confluence property which deals with (countably) infinite sets of arbitrary computations. The results of this chapter are intuitively clear and the reader may skip this chapter at first reading and turn directly to the chapter on fixpoint semantics.

6.1 Basic Concepts and Notation

When we want to show confluence the formulation of the operational semantics causes some problems.

To give a correct treatment of the hiding operator it is necessary to allow an agent to maintain a local state. To simplify the formulation of the operational semantics the local state does not only contain information relevant to the local variable but also a (redundant) copy of the global state. This gives us some technical difficulties since two versions of an agent may differ in that one agent contains an old/incomplete copy of the global state while the other agent contains a fresher copy of the global state. This does not affect the operational behaviour of the two agents (given an appropriate global state) but the syntactic difference between the two agents makes a formulation of a confluence property more complicated.

As an example consider the following configuration.

\[ X = 5 \land \exists_Y Z = 7 : \bot \]

If we perform computation two steps; one with the first part of the conjunction and then with the second part we obtain the following configuration.

\[ X = 5 \land \exists_Y Z = 7 : \bot \rightarrow X = 5 \land \exists_Y Z = 7 : X = 5 \]
\[ \rightarrow X = 5 \land \exists_Y^{(X=5 \land Z=7)} Z = 7 : X = 5 \land Z = 7 \]
If on the other hand we perform the reductions in the opposite order we reach the following situation.

\[
X = 5 \land \exists^2 Z = 7 : \bot \quad \rightarrow \quad X = 5 \land \exists^2 Z = 7 : Z = 7 \\
\quad \rightarrow \quad X = 5 \land \exists^2 Z = 7 : X = 5 \land Z = 7
\]

We see that the only difference between the two final configurations is that \( X = 5 \) occurs as local data in the first but not in the second. This has of course no influence on the external behaviour since the same information (\( X = 5 \)) is available globally.

It follows that two agents may differ in their local data but still exhibit the same external behaviour. To deal with this problem we define a mapping \([-\,]\), over agents which maps an agent to a canonical agent which stores as local data all information which is available globally and visible locally. In the evaluation of \([A],\Gamma\) the constraint \( c \) represents the information which is available globally. The mapping will be used to define an equivalence relation over agents.

**Definition 6.1** For an agent \( A \) and a constraint \( c \Gamma \) let \([A], c\Gamma\) be the agent given by the following rules.

1. \([\exists^X A], c\Gamma\) where \( e = d \lor \exists X c\).
2. \(\prod_{j \in I} A^j, c\Gamma\) where \( \prod_{j \in I} [A^j], c\).
3. \([A], c\Gamma\) if \( A \) is not an existential quantification or a conjunction.

For configurations \( A : c \) and \( B : d \Gamma \) say that \( A : c \equiv B : d \Gamma \) if \( c = d \) and \([A], c\Gamma\) is \([B], d\Gamma\)

\( \square \)

As a motivation for case three in the definition note that tell constraints \( \Gamma \) calls \( \Gamma \) and selections can never store any local data.

It is easy to establish that if \( A : c \longrightarrow B : d \Gamma \) and \( A : c \equiv A' : c' \) then there is some configuration \( B' : d' \) such that \( A' : c' \longrightarrow B' : d' \Gamma \) and \( B : d \equiv B' : d' \).

An abstract configuration \( K \) is an equivalence class of configurations. We will sometimes let the configuration with the canonical agent \([A], c\Gamma\) represent the abstract configuration (equivalence class) containing \( A : c \). For an abstract configuration \( K = [A(s) : c\Gamma] \) let \( store(K) = e \Gamma \) and \( input(K, d) = [A(s) : c \cup d] \). For abstract configurations \( K \) and \( L \) say that \( K \Rightarrow L \) if there are \( (A(s) : c) \in K \) and \( A'(s') : d' \in L \) such that \( A(s) : c \rightarrow A'(s') : c' \). Say that \( K \Rightarrow L \) if \( input(K, c) \Rightarrow LL \Gamma \) for some constraint \( c \).

It is easy to see that if we have a sequence \( K_0 \Rightarrow K_1 \Rightarrow \ldots \) of abstract configurations it is possible to form an input-free computation \( (A_i(s_i) : c_i)_{i \in \omega} \) such that for each abstract configuration \( K_i \) in the sequence there is an \( i \in \omega \) such that \( (A_i(s_i) : c_i) \in K \). A sequence of this type will be referred to as an input-free chain \( \Gamma \) or as a \( \Rightarrow \)-chain.

In the same way \( \Gamma \) given a sequence \( K_0 \Rightarrow K_1 \Rightarrow \ldots \) it is possible to form a computation which contains a configuration for each \( K_i \). A sequence \( K_0 \Rightarrow K_1 \Rightarrow \ldots \) will be referred to as a chain. For a chain \( K_0 \Rightarrow K_1 \Rightarrow \ldots \) the limit \( lim(K_i)_{i \in \omega} = \bigsqcup_{i \in \omega} store(K_i) \).

Also \( \Gamma \) note that if one computation formed from a chain is fair it follows that all computations formed from the chain are fair. We say that a chain is fair if one can form a fair computation from the chain. In the same way \( \Gamma \) we say that a chain is initially fair if one can form an initially fair computation from the chain.
6.2 Finite confluence

The following proposition gives a form of confluence between input and computation steps.

**Proposition 6.2** Suppose $c$ is a constraint and $K$ and $L$ are abstract configurations. If $K \Rightarrow L$ we have $\text{input}(K, c) \Rightarrow \text{input}(L, c)$. If $K \Rightarrow L$ in one step we have $\text{input}(K, c) \Rightarrow \text{input}(L, c)$ in one step.

Note that the proposition does not hold if we instead of considering equivalence classes of configurations consider configurations. It follows that the $\Rightarrow$-relation is transitive as expected.

**Theorem 6.3** (Confluence) If $K \Rightarrow L$ and $K \Rightarrow M$ there is an abstract configuration $N$ such that $L \Rightarrow N$ and $M \Rightarrow N$.

**Proof.** We will only consider the case when $K \Rightarrow L$ and $K \Rightarrow M$ in one computation step. The general case can be treated by a standard induction argument.

Suppose $K = A(s) : c$. The proof is by induction on the agent $A$. The cases where $A$ is a call or a selection or a tell constraint are trivial since there is only one possible reduction step.

Suppose $A = \bigwedge_{j \in I} A^j$. The computation rules for conjunction imply that each computation step with a conjunction is done by performing a computation step with one of the components. It follows that there are $k, l \in I$ such that

$$A^k(\pi_k s) : c \rightarrow B^k(\pi_k s') : c'$$

and

$$A^l(\pi_l s) : c \rightarrow B^l(\pi_l s'') : c''.$$  

(We assume $k \neq l$; the case when $k = l$ can be treated directly using the induction hypothesis.)

To simplify the presentation we re-order the conjunction into a conjunction consisting of three parts $A^k \Gamma A^l \Gamma A^*$ and one agent consisting of all other components in the conjunction. It follows directly from the operational semantics that this re-ordering does not affect the operational behaviour of the agent. Let $A^* = \bigwedge_{j \in I'} A^j \Gamma$ where $I' = I \setminus \{k, l\}$. We assume that $A$ can be written on the form $A^k \land A^l \land A^*$. We have

$$A^k \land A^l \land A^*(s) : c \rightarrow B^k \land A^l \land A^*(s') : c'$$

and

$$A^k \land A^l \land A^*(s) : c \rightarrow A^k \land B^l \land A^*(s'') : c''.$$  

By Proposition 6.2 it follows that

$$A^k(\pi_k s) : c \cup c'' \rightarrow B^k(\pi_k s') : c' \cup c''$$

and

$$A^l(\pi_l s) : c \cup c' \rightarrow B^l(\pi_l s'') : c' \cup c''.$$  

Let $s'''$ be such that $\pi_s s''' = \pi_s \Gamma \pi_k s''' = \pi_k s''' \Gamma$ and $\pi_l s''' = \pi_l s'''$. Also let $c''' = c' \cup c''$. We have

$$B^k \land A^l \land A^*(s') : c' \rightarrow B^k \land B^l \land A^*(s''') : c'''$$

22
Lemma 6.4 Let \((K_i)_{i \in \omega}\) be a chain, and \(c\) a constraint such that \(c \subseteq \text{store}(K_n)\) for some \(n\). Let \(L = \text{input}(K_0, c)\). It follows that \(L \rightarrow K_n\).

Proof. Clearly

\[ L = \text{input}(K_0, c) \rightarrow \text{input}(K_1, c) \rightarrow \ldots \rightarrow \text{input}(K_n, c) = K_n. \]

Intuitively, it should be clear that a fair computation is maximal in the sense that it does as much as possible. We will formalise the notion of a maximal chain and show that all fair chains are indeed maximal.

Definition 6.5 A \(\rightarrow\)-chain \((K_i)_{i \in \omega}\) is maximal if for any abstract configuration \(L\) such that and \(K_i \Rightarrow L\) for some \(i\), there is a \(n \in \omega\) such that \(L \rightarrow K_n\).

Lemma 6.6 Let \((K_i)_{i \in \omega}\) be a fair chain. Then \((K_i)_{i \in \omega}\) is maximal.
Proof. We will only consider the case when $K_0 \Rightarrow L\Gamma$ since if $(K_i)_{i \in \omega}$ is a fair chain it follows that any suffix of the chain is fair. We will also assume that $K_0 \Rightarrow L$ in one step, the general situation can easily be handled by an inductive argument.

The proof is by induction on the agent of $K_0$.

Suppose the agent of $K_0$ is a tell constraint $\Gamma L K_0 = [c(s) : d]$. By the computation rules $L = [c(s) : d \sqcup c]$. Because of fairness there must be an $n$ such that the store of $K_n$ is stronger than $c$. Thus if $K_n = [c(s) : d']$ we have $d' \sqsupseteq d$ and $d' \sqsupseteq d$. It follows that we can go from $L$ to $K_n$ in one input step.

Suppose the agent of $K_0$ is a conjunction. Consider an inner computation of $(K_i)_{i \in \omega}$, $\Gamma$ given by $\text{FACTORIZATION}(K_i)_{i \in \omega} = (K'_i)_{i \in \omega}$. This computation is fair since it is an inner computation of a fair computation. By the computation rules $\Gamma$ must be of the form $\bigwedge_{j \in J} A^{i}(s) : c$. Let $L' = [A^k(\pi_k, s) : c]$. Because of the computation rules $\Gamma$ we have either $K'_0 = L' \Gamma K_0$ or that the step from $K'_0$ to $L'$ is an input step. If $K'_0 = L' \Gamma K_0$ we can apply the induction hypothesis and conclude that $L \Rightarrow K_n \Gamma$ for some $n$. Similarly if the step from $K'_0$ to $L'$ is an input step we apply the induction hypothesis and Proposition 6.4 and find that $L' \Rightarrow K'_n$.

So if $L = \bigwedge_{j \in J} A^{i}(s) : c$ we have $K_n = \bigwedge_{j \in J} B^{j}(s') : d$ and $A^j(\pi_j, s) : d \rightarrow^{*} B^{j}(\pi_j, s') : d$ for all $j$. For a suitable interleaving of the components of the conjunction we have $\bigwedge_{j \in J} A^{i}(s) : d \rightarrow^{*} \bigwedge_{j \in J} B^{j}(s') : d' \Gamma$ and thus $L \Rightarrow K_n$.

If $K_n$ is a selection ($c_1 \Rightarrow A_1, \ldots, c_n \Rightarrow A_n$) alternatives $\Gamma$ and the first element of the oracle of $K_0$ is $k\Gamma$ we must have $1 \leq k \leq n$. It follows that the agent of $L$ is $A_k$. By the fairness requirements there is an $n$ such that $K_n = A_k$. It follows that $L \Rightarrow K_n$.

Suppose the agent of $K_0$ is an existential quantification. Let the chain $(K'_i)_{i \in \omega} = \text{FACTORIZATION}(K_i)_{i \in \omega} \Gamma$ and if $L = \bigwedge_{i \in \omega} A(s) : d$ let $L' = [A(s) : c \sqcup \exists X(d)]$. By the computation rules we have $K'_0 \Rightarrow L'$ in one step and by the induction hypothesis $L' \Rightarrow K'_n \Gamma$ for some $n$. By the computation rules we have $L \Rightarrow K_n$.

Suppose the agent of $K_0$ is a call. It follows that the agent of $L$ is the body of the corresponding procedure. By the fairness assumption there must be an $n$ such that the agent of $K_n$ is the body of the procedure referenced in the call. It follows that $L \Rightarrow K_n$ through an input step.

The converse of Lemma 6.6 does not hold; there are maximal chains which are not fair. Consider for example the configuration

$$K = (1 = 1 \Rightarrow A)(0, s) : c,$$

where $\forall t$'s and $c$ are arbitrary. The single condition of the selection is always true $\Gamma$ but may not be selected since the oracle begins with $0$. Because of the fairness requirement any chain starting with $K$ is not fair $\Gamma$ but the chain $K \Rightarrow K \Rightarrow \ldots$ is maximal.

**Corollary 6.7** Let $(K_i)_{i \in \omega}$ be a fair chain. Let $L$ be such that $K_0 \Rightarrow L$ and $\text{store}(L) \sqsubseteq \lim_{i \in \omega}(K_i)_{i \in \omega}$. There is an $n \in \omega$ such that $L \Rightarrow K_n$.

**Proof.** Let $c = \text{store}(L)$. For $i \in \omega$ let $K'_i = \text{input}(K_i, c)$. Clearly $\Gamma(K'_i)_{i \in \omega}$ is a fair chain. We have $K'_0 \Rightarrow L$. Since $(K'_i)_{i \in \omega}$ is maximal we have $L \Rightarrow K'_n \Gamma$ for some $n$. Choose $m$ such that $m \geq n$ and $\text{store}(K_m) \sqsubseteq c$. By transitivity we have $L \Rightarrow K'_m$. Since $K'_m = K_m$ we have $L \Rightarrow K_m$. 

$\square$
Recall that a computation is initially fair if every agent that occurs in the first configuration and can perform a computation step will eventually do so. Now we consider the situation where two chains have the same initial configuration and one is initially fair. If there is some sequence of input and computation steps from each configuration in the initially fair chain to some configuration in the other chain, it would appear that the second chain should also be initially fair since apparently all computation steps done in the first chain are also done in the second. In the following proposition we verify that this is indeed the case.

**Proposition 6.8** Suppose that we have chains \((K_i)_{i \in \omega}\) and \((L_i)_{i \in \omega}\) such that \(K_0 = L_0\) and for all \(i \in \omega\) there is a \(j \in \omega\) such that \(K_i \rightarrow L_j\). If the chain \((K_i)_{i \in \omega}\) is initially fair it follows that the chain \((L_i)_{i \in \omega}\) is also initially fair.

**Proof.** The proof is by induction of the agent of \(L_0\Gamma\) which of course is also the agent of \(K_0\).

Suppose the agent of \(L_0\) is a tell constraint \(c\). Because of fairness there must be an \(i\) such that \(store(K_i) \supseteq c\). Since \(K_i \rightarrow L_j\Gamma\) for some \(j\) we have \(store(K_i) \subset store(L_j)\Gamma\) thus \(c \subset store(L_j)\) and \((L_i)_{i \in \omega}\) is initially fair.

Suppose the agent of \(L_0\) is a conjunction. Consider an inner computation of \((L_i)_{i \in \omega}\Gamma\) given by \(\text{factor}_i(L_i)_{i \in \omega} = (L'_i)_{i \in \omega}\). If we let \((K'_i)_{i \in \omega} = \text{factor}_i(K_i)_{i \in \omega}\) we know that \((K'_i)_{i \in \omega}\) is initially fair since it is an inner computation of an initially fair computation. Let \(i \in \omega\). By assumption we have \(K_i \rightarrow L_j\Gamma\) for some \(j\). By the computation rules we have \(K'_i \rightarrow L'_j\Gamma\) since \(K'_0 = L'_0\) we can apply the induction hypothesis and conclude that \((L'_i)_{i \in \omega}\) is initially fair. It follows that \((L_i)_{i \in \omega}\) is initially fair.

Suppose the agent of \(L_0\) is a selection with \(n\) alternatives. Suppose also that the oracle of \(L_0\) begins with \(k\Gamma\) where \(1 \leq k \leq n\). Since \(K_0 = L_0\Gamma\) and because of fairness there must be an \(i\) such that the agent of \(K_i\) is the agent of the \(k\)th alternative of the selection. Since \(K_i \rightarrow L_j\Gamma\) for some \(j\) it follows that the agent of \(L_j\) cannot be the agent of \(L_0\). Thus we know that \((L_i)_{i \in \omega}\) will perform at least one computation step \(\Gamma\) and since the only computation step that a selection can perform is to choose the alternative indicated by the oracle we know that \((L_i)_{i \in \omega}\) is initially fair.

If the agent of \(L_0\) is a selection with \(n\) alternatives \(\Gamma\) and the oracle of \(L_0\) begins with \(k\Gamma\) where \(k = 0\) or \(k > n\) it follows directly that \((L_i)_{i \in \omega}\) is initially fair.

Next we consider the case when the agent of \(L_0\) is an existential quantification. Let \((L'_i)_{i \in \omega} = \text{local}(L_i)_{i \in \omega}\) and \(\text{Gamma}(K'_i)_{i \in \omega} = \text{local}(K_i)_{i \in \omega}\). We know that \((K'_i)_{i \in \omega}\) is initially fair since it is an inner computation of an initially fair computation. Let \(i \in \omega\). By assumption we have \(K_i \rightarrow L_j\Gamma\) for some \(j\). Thus \(K_i = [\exists_X A(s) : d]\) and \(L_j = [\exists_X A'(s') : d]\) for a variable \(X\) and appropriately selected agent constraints and oracles \(\Gamma\) and we know that

\[
\exists_X A(s) : d \cup e \rightarrow^* \exists_X A'(s') : d,
\]

for some \(e\). We consider only the case when the reduction is in exactly one step. In this case we have \(\Gamma\) by the computation rules that

\[
A(s) : c \cup \exists_X (d \cup e) \rightarrow A'(s') : c',
\]

and \(d' = d \cup e \cup \exists_X (c')\). We have \(c' \supseteq \exists_X (d \cup e)\) and thus \(c' \supseteq \exists_X (d \cup e) \cup \exists_X (c') = \exists_X (d \cup e \cup \exists_X (c')) = \exists_X (d')\). It follows that \(c' = d' \cup \exists_X (d)\). Since \(K'_i = A(s) :\)
Lemma 6.9 Suppose that we have chains \((K_i)_{i \in \omega}\) and \((L_i)_{i \in \omega}\) such that \(K_0 = L_0\) and \(\lim(K_i)_{i \in \omega} = \lim(L_i)_{i \in \omega}\) and for all \(i \in \omega\) there is a \(j \in \omega\) such that \(K_i \Rightarrow L_j\). If the chain \((K_i)_{i \in \omega}\) is fair it follows that the chain \((L_i)_{i \in \omega}\) is also fair.

Proof. Consider a suffix \((L_i)_{i \geq k}\). We want to show that the suffix is initially fair. We have \(K_0 \Rightarrow L_i\). Thus \(L_i \Rightarrow K_n \Gamma\) for some \(n\). The chain

\[ L_i \Rightarrow K_n \Rightarrow K_{n+1} \Rightarrow \ldots \]

is fair since it has a fair suffix. Proposition 6.8 is applicable since obviously \(L_i \Rightarrow L_j \Gamma\) for some \(j \Gamma\) and by assumption \(K_n \Rightarrow L_j \Gamma\) for some \(j\) and so on. It follows that the chain

\[ L_i \Rightarrow L_{i+1} \Rightarrow L_{i+2} \Rightarrow \ldots \]

is initially fair. Thus \(\Gamma\) every suffix of \((L_i)_{i \in \omega}\) is initially fair and we conclude that \((L_i)_{i \in \omega}\) is fair.

\[\square\]

6.4 Confluence: the general case

Before giving the general confluence property we consider the case when a group of computations can be combined into an input-free computation.

Lemma 6.10 Given an agent \(A\Gamma\) an oracle \(s\) and a constraint \(c\) such that \(\Gamma\) for \(n \in \omega\) \(f_n\) is the functionality of some \(A(s)\)-computation with limit \(c\). Suppose \((\bot \rightarrow c) \subseteq \bigcap_n f_n\). It follows that there is an \(A(s)\)-computation with functionality \((\bot \rightarrow c)\) and limit \(c\).

If \(f_0\) is the functionality of an initially fair \(A(s)\)-computation then there is an initially fair \(A(s)\)-computation with functionality \((\bot \rightarrow c)\) and limit \(c\).

If the computation corresponding to \(f_0\) is fair it follows that there is a fair \(A(s)\)-computation with functionality \((\bot \rightarrow c)\) and limit \(c\).

Proof. Given \(A(s)\) and \(c\) as in the lemma above. For all \(n\) \(\Gamma\) we can construct a chain \(\{K_n\}_{i \in \omega}\) which corresponds the computation which has \(f_n\) as functionality. We assume that \(K_0^n = A(s) : \bot \Gamma\) for all \(n\) (since for any given computation we can form a similar computation where the initial environment is equal to \(\bot \Gamma\) and that for all \(n\) and \(d\) either \(K_n = K_{n+1}^n\) or \(input(K_n^n,d) = K_{n+1}^n \Gamma\) for some constraint \(d\).

We will only consider the case when \(c\) is infinite.
We shall form a chain $L_0 \Rightarrow L_1 \Rightarrow \ldots$ which will have functionality as given by $(\bot \rightarrow c)$ and limit $c$. For each $n$ and $i$ we will have $K_i^n \Rightarrow L_j\Gamma$ for some $j$. We will use the notation $K \xrightarrow{\delta} L$ if $\text{input}(K, \delta) \Rightarrow \text{input}(L, \delta)$.

Let $p$ be a function $p : \omega \rightarrow \omega$ such that $p(i) = n$ infinitely often for each $n$. Let the chain $(L_i)_{i \in \omega}$ be as follows. For each $i \in \omega$ let $c_i = \text{store}(L_i)$.

$$L_0 = K_0^0 (= K_1^1 = K_2^2 = \ldots)$$

$$L_{i+1} = \begin{cases} L_i \ast K_m^{p(i)}, & \text{if } K_m^{p(i)} \xrightarrow{c_i} K_{m+1}^{p(i)} \\ L_i, & \text{otherwise} \end{cases}$$

where $m$ is the largest such that $K_m^{p(i)} \xrightarrow{c_i} L_i$.

There is always at least one $m$ which satisfies the above since we always have $K_0^{p(i)} \xrightarrow{c_i} L_i\Gamma$ furthermore there always a maximal $m$ since for some $m$ we have $\text{store}(K_m^{p(i)}) \equiv c_i$.

Let $d = \bigsqcup_{i \in \omega} \text{store}(L_i)$. We would like $d$ to be equal to $c$. Suppose it is not. It follows that $d \notin f_n\Gamma$ for some $n$. Hence $d$ is the greatest such that $K^n_j \Rightarrow L_i\Gamma$ for some $i$. If $K^n_j \Rightarrow L_j^{n+1}\Gamma$ we know from the way the $L_i$s were selected that $K^{n+1}_j \Rightarrow L_j\Gamma$ for some $j\Gamma$ which contradicts the assumption about $j$. So the step from $K_j^n$ to $K_j^{n+1}$ is an input step. If $\text{store}(K_j^{n+1}) \varsubsetneq d\Gamma$ we would expect $K_j^n \xrightarrow{c_i} K_j^{n+1}\Gamma$ for some $\delta\Gamma$ which again leads to a contradiction. So we conclude that $\text{store}(K_j^{n+1}) \supseteq d\Gamma$ but then $d$ must be a fixpoint of $f_n\Gamma$ and we have arrived at a contradiction. We can conclude that $\bigsqcup_{i \in \omega} \text{store}(L_i) = c$.

Suppose $(K_i^0)_{i \in \omega}$ is initially fair. That $(L_i)_{i \in \omega}$ is $\Gamma\text{fair}$ note that $(K_i^0)_{i \in \omega}$ is $\Gamma\text{fair}$

To establish fairness properties note that for each $i$ there is a $j$ such that $K_i^0 \xrightarrow{c_i} L_i$. If $(K_i^0)_{i \in \omega}$ is initially $\Gamma\text{fair}$ it follows by Proposition 6.8 that $(L_i)_{i \in \omega}$ is initially fair. Similarly if $(K_i^0)_{i \in \omega}$ is $\Gamma\text{fair}$ it follows by Lemma 6.10 that $(L_i)_{i \in \omega}$ is fair.

In the proof of the theorem below we will use the following notation. Given a trace $t$ write $\overline{t}$ for the agent

$$\bigwedge_{i \in \overline{t}} (v(t)_i = v(t)_{i+1})$$

It is easy to see that with the oracle $s = 111\ldots$ and $A = [\overline{t}]$ there is a $A(s)$-computation with limit equal to the limit of $t$ and functionality $g$ such that $g \cap \text{fn} t = (\bot \rightarrow c)$.

**Theorem 6.11** (Generalised confluence) Given an agent $A\Gamma$ an oracle $s$ and a constraint $c$ such that $\Gamma$ for $n \in \omega \Gamma \Gamma_n$ is the functionality of some $A(s)$-computation with limit $c$. Let $t$ be a trace such that $\lim t = c$ and $\text{fn} t \subseteq \bigcup_{n \in \omega} f_n$. Then there is a $A(s)$-computation with trace $\ell\Gamma$ such that $t$ is a subtrace of $\ell\Gamma$.

If there is also an initially $\Gamma\text{fair}$ $A(s)$-computation with limit $c\Gamma$ there is an initially $\Gamma\text{fair}$ $A(s)$-computation with trace $\ell\Gamma$ such that $t$ is a subtrace of $\ell\Gamma$.

If there is a $\Gamma\text{fair}$ $A(s)$-computation with limit $c\Gamma$ it follows that there is a $\Gamma\text{fair}$ $A(s)$-computation with trace $\ell\Gamma$ such that $t$ is a subtrace of $\ell\Gamma$.
Proof. In the case that there is an (initially) fair computation with limit \( c \) we assume that corresponding functionality is \( f_0 \).

Now consider the agent \( A' = A \land [\overline{f}] \). Let \( s' \) be an oracle such that \( \pi_0 s' = s \) and \( \pi_1 s' = 111 \ldots \).

We will construct a family \( \{ f_n \}_{n \in \omega} \) of closure operators such that for each \( n \in \Gamma \) there is an \( A'(s') \)-computation with limit \( c \) and functionality \( f_n \) (and for \( n = 0 \) this computation is (initially) fair when \( f_0 \) corresponds to an (initially) fair computation). Further, we want \( \bigcap_{n \in \omega} f_n' \supseteq (\bot \to c) \).

First let \( f_{n+1} = f_n \Gamma \) for \( n \in \omega \).

Second, note that the agent \( [\overline{f}] \) has a computation with functionality \( g \) which is essentially the inverse of \( f n t \). It is easy to see that we can construct a similar computation of \( A'(s') \) which ignores the agent \( A \). This computation is not fair but this does not matter. Let \( f'_1 = g \).

Last we can take any (initially) fair \( A(s) \)-computation with limit \( c \) and interleave it with the execution of the agent \( [\overline{f}] \) in a suitable manner and obtain an (initially) fair \( A'(s') \)-computation. We need not make any assumptions about the functionality of this computation only that there is such an (initially) fair computation with limit \( c \). Let \( f_0' \) be the functionality of this computation.

Now we have \( \bigcap_{n \in \omega} f_n \supseteq \bigcap_{n > 0} f_n' = g \cap (\bigcap_{n \in \omega} f_n) \supseteq (\bot \to c) \). It follows by Lemma 6.10 that there is a \( A'(s') \)-computation with functionality \( (\bot \to c) \) and limit \( c \).

In other words, there is a trace \( u \in \mathcal{O}_\Gamma[A \land [\overline{f}]] \) such that \( \lim u = c \) and \( \text{fn } u = (\bot \to c) \). It follows that \( u = u_1 \lor u_2 \Gamma \) where \( u_1 \in \mathcal{O}_\Gamma[A] \) and \( u_2 \in \mathcal{O}_\Gamma[\overline{f}] \). Suppose \( d \subseteq c \) such that \( d \in \text{fn } u_1 \). It follows that \( d \not\in \text{fn } u_2 \Gamma \) by the computation rules. Since \( \text{fn } u_2 \subseteq \text{fn } \Gamma \) we have \( d \not\in \text{fn } \Gamma \). By the definition of \( \Gamma \) it follows that \( d \in t \). So \( \text{fn } u_1 \subseteq \text{fn } \Gamma \) i.e. \( \Gamma \text{fn } u_1 \supseteq \text{fn } t \). It follows that \( t \) is a subtrace of \( u_1 \) and we are done.

\[ \square \]

7 Fixpoint semantics

For a given agent \( A \) and oracle \( s \) there is a set \( w \) of limits of all possible fair \( A(s) \)-computations. We will call this set the window of \( A(s) \). This set is convex, and is one component of the domain of the fixpoint semantics we will give in this section.

The confluence theorems tell us that for a given agent \( A \) and oracle \( s \) the set of \( A(s) \)-computations satisfy a number of properties. For example, given a countable set of \( A(s) \)-computations with the same limit \( \Gamma \) we can find an \( A(s) \)-computation which has a functionality which is an upper bound of the functionalities of the computations in the set (provided that the functionality obtained as an upper bound also can be expressed as the functionality of a trace).

We shall see that for each agent \( A \) and oracle \( s \) it is possible to determine a closure operator \( f \) such that each \( A(s) \)-computation has a functionality weaker than \( f \). Further, we will also see that for each trace \( t \) with functionality weaker than \( f \) and a limit which lies in the window of \( A(s) \) there is a fair \( A(s) \)-computation which a functionality stronger or equal to the functionality of the trace \( t \) and limit equal to the limit of \( t \). We will call the closure operator \( f \) which satisfies the above the functionality of \( A(s) \).

Thus the abstract behaviour of an agent \( A \) for a given oracle \( s \) can be described by giving the window and functionality of \( A(s) \). This information is sufficient to
give a compositional semantics \( \Gamma \) but if we want a fixpoint semantics (and we do!) we must find something better since we will find that the existential quantifier in not continuous (or even monotone) under any reasonable ordering.

As the use of oracles does away with non-determinism one might expect that it should be possible to give a fully abstract fixpoint semantics for pairs \( A(s) \) of an agent and an oracle following the idea that \( \Gamma \) for example an agent \( \bigwedge_{j \in I} A_j(s) \) is composed of the agents \( A_j(\pi_j s) \) for \( j \in I \). It is indeed possible to give a fully abstract compositional semantics for agent-oracle pairs using a domain consisting of the functionality and window of an agent-oracle pair. However when we turn to the problem of giving a fully abstract fixpoint semantics for agent-oracle pairs it turns out that the rather complex behaviour of the existential quantifier comes in the way.

**Theorem 7.1** There is no fully abstract fixpoint semantics for agent-oracle pairs.

**Proof.** First note that the semantics of \( \textsf{true}(s) \) (regardless of the choice of the oracle \( s \)) must be the least element of the domain \( \Gamma \) since it has the same behaviour as the (rather meaningless) procedure \( p \) with the definition \( p :: p \Gamma \). It remains completely passive and does not impose any conditions on the input and since the definition of \( p \) corresponds to the functional which is the identity function we can conclude immediately that the semantics of \( p \) must be the least element of the domain.

Now let the agent \( A \) be

\[
A = \textsf{true} \land (X = 3 \Rightarrow \textsf{true} \mid \ldots)
\]

and the agent \( B \)

\[
B = X = 3 \land (X = 3 \Rightarrow \textsf{true} \mid \ldots).
\]

(Only the first part of the selection is shown since the rest is irrelevant to the example. We assume that the two selections in \( A \) and \( B \) are the same.) Let \( s \) be an oracle such that \( \pi_2 s = 1 \). Thus \( A(s) \) and \( B(s) \) are agents which are forced to chose the first alternative in the selection.

By continuity of conjunction we find that \( A(s) \) must be weaker than \( B(s) \) since \( \textsf{true} \) is weaker than \( X = 3 \).

But when we look at \( \exists_X A(s) \) and \( \exists_X B(s) \) we see that the semantics of \( \exists_X B(s) \) is equal to \( \textsf{true} \) since the agent \( \exists_X B(s) \) is completely passive and imposes no conditions on the input.

On the other hand even though the agent \( \exists_X A(s) \) is passive it does impose conditions on the input. Indeed to be able to select the first alternative in the selection it is necessary that the global store (lets call it \( c \)) is such that \( c \Gamma \) when quantified with \( X \) still implies that \( X = 3 \). In other words we must have \( \exists_X c \models (X = 3) \). It is easy to see that the only \( c \) for which this holds is \( c = \top \). In other words the agent \( \exists_X A(s) \) does not generate any output but requires that the store must eventually be equal to \( \top \).

So the agent \( \exists_X B(s) \) is naturally mapped to the least element of the domain \( \Gamma \) while \( \exists_X A(s) \) must be given a distinct and thus stronger semantics. We have arrived at a contradiction and conclude that within the conditions stated above there is no fully abstract fixpoint semantics which gives the semantics of an agent with a given oracle. \( \square \)
The negative result is of some interest in itself since the language under consideration is no longer non-deterministic. Thus this negative result is of a different nature than other published negative results on the existence of fully abstract fix-point semantics [11,2] since they considered non-deterministic languages.

7.1 Hiding

Because of the negative result above we turn to a less abstract domain in which the local state of a computation is included in the semantics. To distinguish between the local and global state we introduce a class of variables which we will call the hidden variables. The idea is that hidden variables are not to be considered part of the external behaviour of an agent.

To deal with the introduction of new hidden variables and with the renaming of hidden variables to prevent clashes between hidden variables of agents in a conjunctions we introduce different kinds of renamings.

The following section presents the appropriate types of renamings and gives some of their properties.

7.1.1 Renamings

Recall that the set of formulas was assumed to contain equality and be closed under conjunction and existential quantification and served as a basis for the definition of constraints. In this section we will return to the formulas and add some more assumptions. In particular we want to be able to talk about renamings i.e. substitutions that replace variables with variables.

Definition 7.2 A renaming $\theta$ is a mapping $\theta : \text{Var} \rightarrow \text{Var}$ over variables.

We extend the set of formulas so that if $\theta$ is a renaming and $\phi$ is a formula then $\theta\phi$ is also a formula.

We assume that a truth assignment $\models$ satisfies the following for a variable assignment $V$ a renaming $\theta$ and a formula $\phi$.

$V \models \theta\phi$ iff $V \circ \theta \models \phi$

As before the set of formulas can be embedded into a Scott domain of constraints using ideal completion. We can extend renaming to constraints according to the following rules.

1. $\theta \uplus R = \bigsqcup_{\delta \in R} \delta d\Gamma$ for directed sets $R \subseteq \mathcal{K}(U)$.

A renaming can thus be seen as a function over variables or over formulas or over constraints. It is easy to see that a renaming is continuous when seen as a function over constraints.

Recall that for a constraint $c$ and an assignment $V$ we write $V \models c$ to indicate that $V \models \phi$ holds for all formulas $\phi \in c$. For a renaming $\theta$ we have $V \models \theta c$ iff $V \circ \theta \models c$.

We will use the notation $\{X \rightarrow Y\}$ for the renaming that maps the variable $X$ to $Y$ and all other variables to themselves. So we have for example $\{X \rightarrow Y\}(X \geq 42) = (Y \geq 42)$. 

30
7.1.2 Injective renamings An injective renaming is a renaming \( \theta \) such that \( \theta \) is injective when seen as a function over variables.

**Proposition 7.3** An injective renaming \( \theta \) is also injective when seen as a function over constraints.

*Proof.* Suppose that \( \theta \) is injective and that \( \theta c = \theta d \) for constraints \( c \) and \( d \). We want to show that \( c = d \).

Suppose that \( \phi \in c \). It follows that \( \theta \phi \in \theta c \). We have \( \phi \in \theta \Delta = \theta c \) thus there is a \( \phi' \in d \) such that \( \theta \phi \leq \theta \phi' \). Let \( V \) be an assignment such that \( V \models \phi' \). Since \( \theta \) is injective there is a renaming \( \theta' \) such that \( \theta' \circ \theta = \text{id} \). It follows that with \( V' = V \circ \theta' \) we have \( V' \models \phi' \) (since \( V' \circ \theta = V \circ \theta' \circ \theta = V \)) and thus \( V' \models \theta \phi' \). Since \( \theta \phi \leq \theta \phi' \) we have \( \phi \models \phi' \) and constraints are assumed to be down-closed sets of formulas we have \( \phi \in d \).

It follows that \( c \subseteq d \) by a symmetric argument we can establish that \( d \subseteq c \) and thus \( c = d \). 

7.1.3 Inverse For an injective renaming \( \theta \) let \( \theta^{-1} \) be the weakest continuous function over constraints such that \( \theta^{-1}(\theta c) \supseteq c \). We have

\[
\theta^{-1} c = \bigsqcup \{ d \mid \theta d \subseteq c \}.
\]

It follows that \( \theta^{-1}(\theta c) = c \) and \( \theta(\theta^{-1} c) \supseteq c \).

An injective renaming is a renaming \( \theta \) such that \( \theta \) is injective when seen as a function over variables. For an injective renaming \( \theta \) we have \( \theta^{-1} \circ \theta = \text{id} \).

**Proposition 7.4** Let \( \theta \) be an injective renaming. We have \( c \subseteq \theta^{-1} d \) iff \( \theta c \subseteq d \) for constraints \( c, d \).

*Proof.* If \( c \subseteq \theta^{-1} d \) we have \( \theta c \subseteq \theta(\theta^{-1} d) \subseteq d \). If \( \theta c \subseteq d \) we have \( \theta^{-1}(\theta c) \supseteq \theta^{-1} d \) since \( \theta^{-1}(\theta c) = \theta c \). It follows that \( c \subseteq \theta^{-1} d \). 

7.1.4 Hidden variables In the following text we will assume the existence of a set \( H \) of hidden variables. We assume that hidden variables do not occur in agent’s programs, traces or computations. The hidden variables will be used to represent the internal state of a computation. (It is perhaps worthwhile to point out that hidden variables and constraints involving hidden variables are not in any way different from other variables and constraints. The only difference is the assumption that hidden variables are not to be used in agent’s programs and so on.) The variables which are not hidden will sometimes be referred to as visible variables.

7.1.5 Operations on constraints with hidden variables Basically we need three types of operations on hidden variables. We will write \( \exists_H c \) for the existential quantification of all variables occurring in the constraint \( c \).

The second operation is the renaming \( \text{new}_X \Gamma \) for a visible variable \( X \). We will use \( \text{new}_X \) to model the existential quantification of variables.

Let \( \text{new}_X \) be an injective renaming which

1. maps \( X \) to a hidden variable \( \Gamma \) and
2. maps every visible variable distinct from \( X \) to itself \( \Gamma \) and
3. maps every hidden variable to a hidden variable.

Since \( \text{new}_X \) is assumed to be injective there is an inverse \( \text{new}_X^{-1} \) which satisfies \( \text{new}_X^{-1} \circ \text{new}_X = \text{id} \) and \( \text{new}_X \circ \text{new}_X^{-1} = \text{id} \).

**Proposition 7.5** \( \exists_H \circ \text{new}_X = \exists_H \circ \exists_X \)

**Proof.** We will show that given a constraint \( c \) and a variable assignment \( VT \) we have \( V \models \exists_H(\exists_X c) \iff V \models \exists_H(\text{new}_X c) \).

First note that \( V \models \exists_H(\exists_X c) \iff \) there is an assignment \( V' \) such that \( V' \models c \) and \( V'(Y) = V(Y) \) for visible variables \( Y \) distinct from \( X \).

Second, \( V \models \exists_H(\text{new}_X c) \iff \) there is an assignment \( V'' \) such that \( V'' \models \text{new}_X c \) and \( V(Y) = V''(Y) \Gamma \) for visible variables \( Y \). By the definition of \( \models \) we have \( V'' \models \text{new}_X c \iff V'' \circ \text{new}_X \models c \).

It is now easy to see that \( V'' \circ \text{new}_X \) satisfies the condition for \( V' \) above. Thus if \( V \models \exists_H(\text{new}_X c) \) we also have \( V \models \exists_H(\exists_X c) \).

In the other direction note that if \( V \models \exists_H(\exists_X c) \) holds (and we have \( V' \models c \)) we can construct an assignment \( V'' \) such that \( V''(Y) = V(Y) \) for visible variables \( YT \) and \( V''(\text{new}_X X) = V'(X) \). An assignment \( V'' \) that satisfies these conditions also satisfies \( V'' \circ \text{new}_X = V'' \Gamma \) and it follows that \( V \models \exists_H(\text{new}_X c) \).

It follows immediately that \( \exists_H(\text{new}_X c) = \exists_H(\exists_X c) \) for arbitrary constraints \( c \). \( \boxdot \)

**Proposition 7.6** Let \( c \) be a constraint independent of hidden variables. It follows that \( \text{new}_X(\exists_X c) \subseteq c \).

**Proof.** Suppose \( V \models c \). Let \( V' = V \circ \text{new}_X \). Since \( V'(Y) = V(Y) \) for all visible variables \( Y \) except \( XT \) and \( c \) does not depend on hidden variables we have \( V' \models \exists_X(\exists_X c) \Gamma \). Thus \( V \models \exists_H(\exists_X c) \Gamma \) which implies that \( V \models \text{new}_X(\exists_X c) \).

**Proposition 7.7** Let \( c \) be a constraint independent of hidden variables. Let \( X \) be a visible variable. It follows that \( \text{new}_X^{-1}(\exists_X c) = \exists_X c \).

**Proof.** \((\Rightarrow)\) By Proposition 7.6, \( \text{new}_X(\exists_X c) \subseteq c \). By applying \( \text{new}_X^{-1} \) on both sides we find that \( \exists_X c \subseteq \text{new}_X^{-1}(\exists_X c) \).

\((\Leftarrow)\) Note that \( \text{new}_X^{-1}(\exists_X c) = \bigcup \{ d \mid \text{new}_X d \subseteq x \} \). To show that \( \exists_X c \subseteq \text{new}_X^{-1}(\exists_X c) \) it is sufficient to show that for all \( d \) such that \( \text{new}_X d \subseteq c \) we have \( d \subseteq \exists_X c \).

Let \( d \) be such that \( \text{new}_X d \subseteq c \). Let \( V \) be a variable assignment such that \( V \models \exists_X c \). There is a variable assignment \( V' \) such that \( V' \models c \) and \( V'(Y) = V(Y) \Gamma \) for all variables \( Y \neq X \). Let \( V'' \) be an assignment such that \( V''(\text{new}_X Y) = V(Y) \Gamma \) for all variables \( YT \) and \( V''(X) = V(X) \). Clearly \( V''(Y) = V'(Y) \) for all visible variables \( Y \). Since \( c \) by assumption does not depend on visible variables it follows that \( V'' \models c \). Thus \( V'' \models \text{new}_X d \Gamma \) and \( V'' \circ \text{new}_X \models d \). Since \( V'' \circ \text{new}_X = VT \) we have \( V \models d \). So \( d \subseteq \exists_X d \). \( \boxdot \)

When considering the semantics of a parallel conjunction \( \bigwedge_{j \in I} A_j \) we need a way to keep the hidden variables of the agents in the conjunction from interfering with each other. To accomplish this we assume that for each parallel conjunction \( \bigwedge_{j \in I} A_j \) there is a family of injective renaming (called projections) \( \{ \theta_j \}_{j \in I} \) such that (writing \( \theta_j H \) for \( \{ \theta_j X \mid X \in H \} \))
1. \((\theta_j \Gamma) \cap (\theta_j \Gamma) = \emptyset \) for \( j \neq j' \) and 
2. \( \theta_j \chi = X \Gamma \) for visible variables \( X \).

### 7.1.6 Applying renamings on sets and closure operators

Injective renamings can be generalized to sets and closure operators.

For an injective renaming \( \theta \Gamma \) and a set of constraints \( S \), let 
\[ \theta S = \{ c \mid \theta^{-1} c \in S \}. \]

#### Proposition 7.8

For a closure operator \( f \Gamma \) and an injective renaming \( \theta \Gamma \) we have 
\( c \in \theta f \) iff \( c \) is a fixpoint of \( \theta \circ f \circ \theta^{-1} \cup \text{id} \).

**Proof.**

(\( \Rightarrow \)) Suppose that \( c \) is a fixpoint of \( \theta \circ f \circ \theta^{-1} \cup \text{id} \). It follows that 
\( (\theta \circ f \circ \theta^{-1}) c \subseteq c \). Thus \( \theta^{-1}((\theta \circ f \circ \theta^{-1}) c) \subseteq \theta^{-1} \text{id} \Gamma \) and \( f(\theta^{-1} c) \subseteq \theta^{-1} c \). Since \( f \) is a closure operator it follows that \( \theta^{-1} c \) is a fixpoint of \( f \) and thus \( c \in \theta f \).

(\( \Leftarrow \)) Suppose that \( c \in \theta f \). It follows that \( \theta^{-1} c = f(\theta^{-1} c) \Gamma \) and thus \( \theta(f(\theta^{-1} c)) \subseteq \theta(\theta^{-1} c) \subseteq c \). We have \( \theta(f(\theta^{-1} c)) \cup c = \theta \Gamma c \) and thus \( c \) is a fixpoint of \( \theta \circ f \circ \theta^{-1} \cup \text{id} \).

#### Proposition 7.9

Let \( S \) be a set of constraints and \( X \) a variable. It follows that 
\( E_H(E_X S) = E_H(\text{new}_X S) \).

**Proof.**

(\( \supseteq \)) Let \( c \in E_H(\text{new}_X S) \). There is a constraint \( d \) such that \( \exists_H d = \exists_H c \) and \( \text{new}_X^{-1} d \in S \). Let \( e = \text{new}_X^{-1} d \). To prove that \( c \in E_H(E_X S) \) it is sufficient to find a constraint \( d \) such that \( \exists_H c = \exists_H d \) and \( \exists_X d = \exists_X e \). Let \( d = \exists_H \cup \exists_X e \). First note that \( \exists_H d = \exists_H(\exists_H \cup \exists_X e) = \exists_H \cup \exists_H(\exists_X e) \). Since \( \exists_H(\exists_X e) = \exists_H(\text{new}_X e) \subseteq \exists_H d = \exists_H c \) we have \( \exists_H c = \exists_H d \). Second we have \( \exists_X d = \exists_X(\exists_X e) \). Since
\[
\exists_H(\exists_X e) = \exists_X(\exists_H d)
= \exists_X(\exists_X(\exists_H d))
= \exists_X(\text{new}_X^{-1}(\exists_H d))
\subseteq \exists_X(\text{new}_X^{-1} d)
= \exists_X e,
\]
it follows that \( \exists_X d = \exists_X e \).

(\( \subseteq \)) Let \( c \in E_H(E_X S) \). There is a constraint \( e \in S \) such that \( \exists_H(\exists_X e) = \exists_H(\exists_X c) \). We would like to find a constraint \( d \) such that \( \exists_H d = \exists_H c \) and \( \text{new}_X^{-1} d \in S \). Let \( d = \exists_H \cup \text{new}_X \). First we see that \( \exists_H d = \exists_H \cup \exists_H(\text{new}_X e) \). But \( \exists_H(\text{new}_X e) = \exists_H(\exists_X e) = \exists_H(\exists_X c) \subseteq \exists_H \Gamma \) so we can conclude that \( \exists_H d = \exists_H c \). Second applying the function \( \text{new}_X^{-1} \) gives us \( \text{new}_X^{-1} d = \text{new}_X^{-1}(\exists_H d) \cup \text{new}_X^{-1}(\text{new}_X e) = \text{new}_X^{-1}(\exists_X d) \cup e \). We see that \( \text{new}_X^{-1}(\exists_X d) = \exists_X(\exists_H d) = \exists_H(\exists_X e) \subseteq e \Gamma \) thus \( \text{new}_X^{-1} d = e \).

#### Proposition 7.10

Let \( \theta_j \) be a projection and \( c \) a constraint that does not depend on hidden variables. It follows that \( \theta_j c = c \) and \( \theta_j^{-1} c = c \).

**Proof.**

Suppose that \( V \models c \Gamma \) for an assignment \( V \). Since \( \theta_j(X) = X \Gamma \) for visible variables \( X \) we have \( V \circ \theta_j \models c \) and thus \( V \models \theta_j c \). The proof that \( V \models \theta_j c \) implies \( V \models c \) is similar.

To show \( \theta_j^{-1} c = c \Gamma \) we use the result we just obtained \( \theta_j c = c \Gamma \) and apply the inverse projection \( \theta_j^{-1} \) to both sides.
Proposition 7.11 Let \( \theta_j \Gamma \theta_k \) be projections with \( j \neq k \). It follows that \( \theta_j \circ \theta_k = \exists_H \).

Proof. \((\subseteq)\) Note that if \( \exists_H c = d \) we have \( \theta_j(\theta_k d) = \theta_j d = d \). Thus \( (\theta_j \circ \theta_k) c \subseteq (\theta_j \circ \theta_k) d = \exists_H c \).

\((\supseteq)\) In general if \( V \models \theta_j c \) we have \( V \circ \theta_j \models c \). Since \( V(X) = (V \circ \theta_j)X \) for visible variables \( X \) we have \( V \models \exists_H c \). (make this a separate proposition??) Thus \( \theta c \models \exists_H c \). It follows immediately that \( \theta_j \circ \theta_k \models \exists_H \circ \exists_H = \exists_H \). \( \square \)

7.2 Trace bundles

The idea is that given an agent \( A \) and an oracle \( s \) the functionality of \( A(s) \) should be a closure operator which corresponds to the maximum functionality of computations of \( A(s) \). In the same way the window associated to \( A(s) \) is the set of constraints which are limits of some fair \( A(s) \)-computation.

Let \( cl \) be the lattice of closure operators over \( \Gamma \Lambda \) and let \( w \) be the lattice of windows over \( \Gamma \Lambda \) with elements \( \varphi(\Lambda) \) ordered by reverse inclusion.

Let \( \text{bundle} \) the \textit{trace bundles} be the set of pairs \( \langle f, w \rangle \) in \( cl \times w \) such that \( \exists_X f = f \Gamma \Lambda \) and \( \exists_X w = w \Lambda \) for any hidden variable.

For a trace bundle \( \langle f, w \rangle \) let \( F \langle f, w \rangle = f \Gamma \Lambda \) and \( W \langle f, w \rangle = w \). Let \( \subseteq \subseteq \text{bundle} \times \text{bundle} \) be defined so that \( \langle f, w \rangle \subseteq \langle f', w' \rangle \) iff \( f \subseteq f' \Gamma \Lambda \) and \( w \supseteq w' \).

Under this ordering \( \text{bundle} \) forms a complete lattice \( \Gamma \wedge = \langle \perp, \top, 0 \rangle \Gamma \Lambda = \langle \perp \rightarrow \top, 0 \rangle \Gamma \Lambda \) and

\[
\langle f_1, w_1 \rangle \cup \langle f_2, w_2 \rangle = \langle f_1 \cap f_2, w_1 \cap w_2 \rangle.
\]

Each trace bundle \( \langle f, w \rangle \) determines a set of traces

\[
\{ t \mid \text{fn } t \subseteq \exists_H(f), \lim t \in (f \circ \exists_H)^{-1} w \cap \exists_H(w) \}.
\]

7.3 Basic Operations

7.3.1 Tell constraints First \( \Gamma \) to give the semantics of a tell constraint \( c \) we define \( \langle c \rangle \) to be the trace bundle with a functionality which adds \( c \) to the store and a window that makes sure that \( c \) is in the store. Let

\[
\langle c \rangle = \langle \perp \rightarrow c, \{ c \} \rangle.
\]

7.3.2 Parallel Composition Given a family of trace bundles \( \{ (f_j, w_j) \}_{j \in \Lambda} \) we can obtain the parallel composition of the trace bundles by simply taking their least upper bound but since we want to keep the local variables of each agent apart we must first apply the projections to rename the local variables. Thus the parallel composition is found using the following expression.

\[
\left\langle \bigcap_{j \in \Lambda} \theta_j f_j, \bigcap_{j \in \Lambda} \theta_j w_j \right\rangle
\]
7.3.3 Selections First note that a selection has two types of behaviour; the first is when one of the conditions becomes satisfied by the store and the corresponding alternative is selected. The other type of behaviour is when no condition ever becomes true. In this case the selection remains passive throughout the computation. It is convenient to treat these two behaviours separately.

First consider an alternative consisting of an ask constraint $c$ and an agent $A$.

First we define select : $\mathcal{U} \to \text{BUNDLE} \to \text{BUNDLE}$ which for a given constraint $c$ takes a trace bundle and returns a trace bundle which does not generate any output until $c$ is satisfied and which requires that $c$ is eventually satisfied.

$$\text{select } c \langle f, w \rangle = \langle c \to f, \{c\}^u \cap w \rangle$$

Now it is easy to give the definition of select as a function

$$\text{select } : \mathcal{U} \to A \to A.$$ 

We lift the definition of select from BUNDLE to $A$ by

$$\text{select } c a s = \text{select } (a s),$$

for $a \in A$ $s \in \text{ORACLE}$.

For the case when in a selection no alternative is ever chosen we define a function

$$\text{unless } : \mathcal{U}^n \to \text{BUNDLE}$$

for $n \geq 0$ as follows. Given constraints $c_1, \ldots, c_n$ let

$$\text{unless}(c_1, \ldots, c_n) = \left(\text{id}, \bigcap_{1 \leq k \leq n} \mathcal{U} \setminus \{c_k\}^u\right).$$

7.3.4 Existential quantification the function $E_X : \text{BUNDLE} \to \text{BUNDLE}$ which gives the trace bundle for $\exists_X A(s)\Gamma$ given the trace bundle for $A(s)$.

Let

$$E_X \langle f, w \rangle = \langle \text{new}_X f, \text{new}_X w, \rangle.$$ 

7.4 Fixpoint Semantics

The oracle fixpoint semantics is given in Figure 3.

8 Correctness

We would like to prove a direct correspondence between the fixpoint semantics and the operational semantics.

First we define abstraction operators $\alpha : \text{BUNDLE} \to \wp(\text{TRACE})$ and $\alpha : (\text{ORACLE} \to \text{BUNDLE}) \to \wp(\text{TRACE})$.

**Definition 8.1** For $\langle f, w \rangle \in \text{BUNDLE}$ let $\alpha \langle f, w \rangle$ be the set

$$\{t \mid \text{fnt} \subseteq \exists_H(f), \lim t \in (f \circ \exists_H)^{-1}w \cap \exists_H(w)\}.$$ 

For $a \in A\Gamma$ let

$$\alpha a = \{\alpha \langle f, w \rangle \mid s \in \text{ORACLE}, a s = \langle f, w \rangle\}.$$ 

□
Definition of \( E[A] : A^N \rightarrow A \)

\[
E[c] \sigma s = \langle c \rangle \\
E[\bigwedge_{j \in I} A^j] \sigma s = \left\langle \bigcap_{j \in I} \theta_j f_j, \bigcap_{j \in I} \theta_j w_j \right\rangle \\
E[\exists X A] \sigma s = E_X (E[A] \sigma s) \\
E[p(X)] \sigma s = \{ \Delta \rightarrow X \}(\sigma s)
\]

Definition of \( P[\Pi] : A^N \rightarrow A^N \)

\[
P[\Pi] \sigma p s = \{ Y \rightarrow \Delta \}(\sigma p s),
\]

where for each \( p \in \mathbb{N} \) the definition in \( \Pi \) is assumed to be of the form \( p(Y) :: \Gamma \) for some variable \( Y \) and some agent \( A \)

Figure 3: The oracle fixpoint semantics

The following proposition expresses a simple relationship between the functionality and window of an agent \( \Gamma \) for a given oracle.

**Proposition 8.2** Given \( \sigma \) such that \( \sigma \) is the least fixpoint of \( P[\Pi] \Gamma \) for some program \( \Pi \), it holds that \( w \subseteq f \Gamma \) where \( \langle f, w \rangle \in E[\Pi] \sigma s \).

It is straightforward to verify the proposition by examination of the semantic functions.

### 8.1 Soundness

First we show that any trace of the operational semantics falls into the set of traces given by the oracle semantics \( \Gamma \) i.e. \( \Gamma \) that the operational semantics is sound with respect to the oracle semantics.

**Lemma 8.3** Let \( t \) be a trace such that \( t \in O[\Pi](s) \). We have \( \text{fn}(t) \subseteq E[H] \Gamma \) where \( f = F(E[\Pi] \sigma s) \Gamma \) and \( \sigma \) is the least fixpoint of \( P[\Pi] \).

**Proof.** We will prove the following by induction on pairs \( (k, A) \Gamma \) under the lexical ordering. Let \( f = F(E[\Pi] \sigma s) \). For any \( A(s) \)-computation \( (A_i : c_i)_{i \in \omega} \Gamma \) when \( A_k : c_k \rightarrow A_{k+1} : c_{k+1} \Gamma \) we have \( c_{k+1} \subseteq (E[H] \Gamma) c_k \).

If \( A \) is a tell constraint \( \Gamma \), \( A \Gamma = d \) for some constraint \( d \Gamma \) it follows immediately that a computation can have a functionality which is at most \( (\bot \rightarrow d) \).

Suppose \( A = \bigwedge_{j \in I} A^j \). Let \( \langle f_j, w_j \rangle = E[A^j] \sigma (\pi_j s) \Gamma \) for \( j \in I \). By the computation rules \( A_i = \bigwedge_{j \in I} A^j_i \Gamma \) for all \( i \in \omega \Gamma \) and there is a \( j \in I \) such that \( A^j_i : c_k \rightarrow A^j_{k+1} : c_{k+1} \). By the induction hypothesis we have \( c_{k+1} \subseteq (E[H] \Gamma) c_k \). 

---

36
Since \( f \supseteq \theta_j f_j \Gamma \) and \( E_H(\theta_j f_j) = E_H f_j \Gamma \) we have \( c_{k+1} \subseteq (E_H f_j) c_k = (E_H(\theta_j f_j))c_k \subseteq (E_H f) c_k \).

If \( A \) is a selection \( \Gamma \) it follows that there must be some position \( i \) at which \( A \) is reduced to one of its branches. Consider the computation beginning at position \( i \).

We can immediately apply the induction hypothesis.

Suppose \( A = \exists X A' \). Let \( f' = F(E[A']) \sigma s \). For each \( i \in \omega \Gamma \) we know that \( A_i = \exists X A'_i \). By the computation rules we have \( A'_k : d_k \cup \exists X(c_k) \rightarrow A'_{k+1} : d_{k+1} \Gamma \) where \( c_{k+1} = c_k \cup \exists X(d_{k+1}) \). For \( i \leq k \Gamma \) we have by the induction hypothesis that \( d_{i+1} \subseteq f'(d_i \cup \exists X(c_i)) \). We can prove by an inductive argument that \( d_{i+1} \subseteq (f' \circ \exists X)c_k \) for all \( i \leq k \Gamma \). First \( d_0 = \perp \Gamma \) by the assumption that the local data in an initial configuration is \( \perp \). Suppose \( d_i \subseteq (f' \circ \exists X)c_k \). Now we have \( \Gamma \) when \( i \leq k \Gamma \) that

\[
d_{i+1} \subseteq f'(d_i \cup \exists X(c_i))
= (f' \circ \exists X)c_k
= (f' \circ \exists X)c_k.
\]

In particular we have \( d_{k+1} \subseteq (f' \circ \exists X)c_k \). It follows that \( c_{k+1} \subseteq c_k \cup (\exists X \circ f' \circ \exists X)c_k = f c_k \). By monotonicity we have \( \exists H(c_{k+1}) \subseteq \exists H(f c_k) \Gamma \) and since \( c_k \) and \( c_{k+1} \) are independent of variables in \( H \), \( c_{k+1} \subseteq \exists H(f c_k) = \exists H(f(\exists X c_k)) \subseteq (E_H f) c_k \).

If \( A = p(X) \Gamma \) for some procedure name \( p \) and variable \( X \Gamma \) and the definition of \( p \) is of the form \( p(Y)::A' \Gamma \) it follows from the computation rules that the first computation step leads a configuration \( A_i = \{ \Delta \rightarrow X \}(\{ Y \rightarrow \Delta \}, A') \). (Clearly \( i > 0 \).) The fixpoint semantics gives us that

\[
E[p(X)] \sigma s = \{ \Delta \rightarrow X \}(\sigma ps)
= \{ \Delta \rightarrow X \}(\{ Y \rightarrow \Delta \}(E[A'] \sigma s))
= E[A] \sigma s.
\]

In particular \( f = E_H(F(E[A] \sigma s)) = E_H(F(E[A_i] \sigma s)) \). It follows that if we consider the computation that starts with the agent \( A_i \) we see immediately that \( f(c_k) \supseteq c_{k+1} \).

\begin{lemma}
Let \( t \) be a trace such that \( t \in \mathcal{O}_H[A] \). We have \( \lim(t) \in E_H w \Gamma \) where \( \sigma \) is the least fixpoint of \( P(\Pi) \Gamma \) and \( w = W(E[A] \sigma s) \).
\end{lemma}

\textbf{Proof.} Let \( \sigma_0 = \lambda s. \lambda \rho. (\text{id}, \Pi) \sigma_0 \Gamma \sigma_{n+1} = P(\Pi) \sigma_n \Gamma \) for \( n \geq 0 \). Let \( w_n = W(E[A]\sigma_n) \Gamma \) for \( n \geq 0 \). Note that \( w = \bigcap_{n \in \omega} w_n \Gamma \) so if \( \lim(t) \notin w \Gamma \) it follows that \( \lim(t) \notin w \Gamma \Gamma \) for some \( n \).

We will prove the following \( \Gamma \) by induction on pairs \( (n, A) \Gamma \) under the lexical ordering. If \( t \in \mathcal{O}_H[A] \Gamma \) it follows that \( \lim(t) \in W(E[A] \sigma_n) \Gamma \). The lemma follows immediately.

If \( A \) is a tell constraint \( c \) it follows immediately from the fairness requirements that any trace \( t \in \mathcal{O}_H[A] \) must have \( \lim(t) \supseteq c \Gamma \), i.e. \( \lim(t) \in \{ c \}^w = w \Gamma \) and since \( \lim(t) \) is independent of hidden variables \( \lim(t) \in E_H w \).

Suppose \( A = \bigwedge_{j \in I} A^j \). It follows by Lemma 5.3 that there are traces \( t_j \in \mathcal{O}_H[A^j] \) such that \( t = \bigvee_{j \in I} t_j \Gamma \). By the induction hypothesis we have \( \lim(t_j) \in E_H(w^j) \Gamma \) where \( w^j = W(E[A^j] \sigma_n) \). Since \( \lim(t) = \lim(t_j) \Gamma \) for all \( j \in I \) \( \lim(t) \in \bigcap_{j \in I} E_H(w^j) = E_H(w) = E_H w \).

37
Suppose $A$ is a selection $(d_1 \Rightarrow A_0 \mid \ldots \mid d_m \Rightarrow A_m)\Gamma$ and $s = k.s'$. Suppose $1 \leq k \leq m$. Let $(A_i, c_i)_{i \in \omega}$ be the computation corresponding to the trace $t$.

By the fairness requirement there is some $l > 0$ such that $A_l = A_k\Gamma, i.e.\,\Gamma$ the $k$th alternative must eventually be selected. By the computation rules we find that this cannot happen unless $c_l \supseteq d_k\Gamma$ for some $l' \leq l$. Now consider the $A_k(s')$-computation $(A_i, c_i)_{i \in \omega}$. By the induction hypothesis we know that the limit of this computation $\Gamma^{l_{\omega}}$ lies in $E_H(W(E[A_k]_{\sigma_n})^s)$. It follows that $\lim t \in E_{Hw}$.

Suppose $A$ is a selection $(d_1 \Rightarrow A_0 \mid \ldots \mid d_m \Rightarrow A_m)$ and $s = 0.s'$. By the fairness requirement it holds for each store $c_i$ that $c_i \supseteq d_k\Gamma$ for $k \leq m$. Thus we have $\lim t = \cup_{i \in \omega} c_i \supseteq d_k\Gamma$ for $k \leq m\Gamma$ and it follows that $\lim t \in W(E[A]_{\sigma_n})$.

Suppose $A = 3^{X}A'$. We know that $t$ is of the form $\langle 3^{d_i}A'_i : c_i \rangle_{i \in \omega}$ and that $t' \in \mathcal{O}_\Pi[A'_{\sigma}]$ where $t' = (A'_i : d_i \cup (3^{X}c_i))_{i \in \omega}$. It follows by the assumption on $A$ that $d_i \cup (3^{X}c_i) \in E_{Hw_n}\Gamma$ where $w_n = W(E[A']_{\sigma_n})\Gamma$ and $c = \cup_{i \in \omega} c_i \Gamma$ and $d = \cup_{i \in \omega} d_i$. By the computation rules we know that $c \supseteq 3^{X}d$. It follows that $3^{X}c \subseteq 3^{X}(c \cup 3^{X}d) = 3^{X}(d \cup 3^{X}c) \in E_{Hw_n}\Gamma$. Thus $c \in E_X(E_{Hw'_n}) = E_{H}(E_X w_n) = E_{H}(new c w_n) = E_{H}(w_n)$.

Suppose $A = p(X)$. There is by the fairness requirement a configuration $A_i : c_i\Gamma$ where $A_i = \{\Delta \rightarrow X\}((Y \rightarrow \Delta)A') \Gamma$ where the definition of $p$ is of the form $p(Y) : A'$. It is easy to see that $E[A]_{\sigma s} = E[A_i]_{\sigma s}$. By the induction hypothesis we know that $c \in \{\Delta \rightarrow X\}((Y \rightarrow \Delta)w')\Gamma$ where $w' = W(E[A']_{\sigma s})$. Since $w = \{\Delta \rightarrow X\}((Y \rightarrow \Delta)w$ we have immediately that $c \in w$ and since $c$ does not depend on any hidden variables it that $c \in E_{Hw}$.

\[\square\]

### 8.2 Completeness

If a trace is given by the oracle semantics we would like to show that the trace for a stronger trace can be obtained from the operational semantics. We state this in the following theorem.

**Theorem 8.5** Let $t \in o(E[A]_{\sigma s}\Gamma)$ for an agent $A\Gamma$ and oracle $s$. There is a trace $t' \in \mathcal{O}_\Pi[A(s')\Gamma]$ such that $t$ is a subtrace of $t'$.

This section is devoted to the proof of the theorem. To avoid repetitions we will assume a program $\Pi\Gamma$ an agent $A\Gamma$ and oracle $s$. We also assume that $\sigma_0 = \bot\Gamma$ and $\sigma_{n+1} = \Pi\Pi[\sigma_n]_\Gamma$ for $n \in \omega\Gamma$ and $\sigma = \cup_{i \in \omega} \sigma_n$.

We need to define some concepts. Note that limits of traces are limits of $\omega$-chains of compact elements. Since we will reason about constraints with this property it is appropriate to give the concept a name and state some of its properties.

Given an algebraic lattice $L\Gamma$ say that $x \in L$ is $\omega$-approximable if there is an $\omega$-chain $x_0, x_1, \ldots$ in $K(L)$ such that $\cup_{i \in \omega} x_i = x$.

Any finite element of $L$ is $\omega$-approximable of course. Also note that if $x_0, x_1$ is a chain of $\omega$-approximable elements it follows that $\cup_{i \in \omega} x_i$ is $\omega$-approximable. Also given algebraic lattices $L_1$ and $L_2\Gamma$ and a function $f : L_1 \rightarrow L_2\Gamma$ which is $\omega$-approximable in the space of continuous functions from $L_1$ to $L_2\Gamma$ it holds that $f(x)$ is $\omega$-approximable for any $\omega$-approximable $x \in L_1$.

**Proposition 8.6** Let $A$ be an agent $\Gamma\sigma$ am $\omega$-approximable environment and $s$ an oracle. Then $F(E[A]_{\sigma s})$ is $\omega$-approximable.
For a program $\Pi\Gamma$ the function $P[\Pi]$ is $\omega$-approximable. The least fixpoint of $P[\Pi]$ is $\omega$-approximable.

**Proposition 8.7** Let $c$ and $d$ be $\omega$-approximable constraints. Let $f$ be a closure operator over constraints such that $f \supseteq (c \to d)$. Then there is a trace $t$ such that $\lim t = d$ and $(c \to d) \subseteq \text{fn}\ t \subseteq f$.

**Proof.** We have $c = \bigcup_{i \in \omega} e_i$ and $d = \bigcup_{i \in \omega} d_i\Gamma$ for $e_0, \ldots$ and $d_0, \ldots$ finite constraints.

Construct the sequence $e_0, e_1, \ldots$ as follows.

Let $e_0 = \bot$. Let $e_{2i} = c_i \cup e_{2i-1}\Gamma$ for $i > 0$. Let $e_{2i+1} = e_{2i} \cup d_j\Gamma$ where $j$ is the greatest such that $j \leq i$ and $d_j \subseteq f(e_{2i})$. Let $t$ be the trace with $v(t) = (e_i)_{i \in \omega}$ and $v(t) = \{i \mid i \text{ is odd}\}$.

We want to show that $\text{fn}\ t \supseteq (c \to d)$. It is sufficient to show that $(\text{fn}\ t) \cap (\bot \to c) \supseteq (\bot \to d)$. Clearly $(\text{fn}\ t) \cap (\bot \to c) \supseteq (\bot \to \lim t)$. If we can show that $\lim t \not\supseteq d$ we are done.

Suppose that $\lim t \not\supseteq d$. There is a least $j$ such that $\lim t \not\supseteq d_j$. We have $f(c) \supseteq d \supseteq d_j$ and thus a least $k$ such that $f(e_k) \supseteq d_j$. Let $i$ be the maximum of $j$ and $k$. We have $f(e_i) \supseteq d_i\Gamma$ and thus $f(e_{2i}) \supseteq d_i\Gamma$ and by the construction above $e_{2i+1} \supseteq d_i$. We have arrived at a contradiction and conclude that $\text{fn}\ t \supseteq (c \to d)$. $\Box$

As a step toward the proof of completeness $\Gamma$ we show the following proposition which essentially implies that if the trace is given by the oracle semantics we can construct traces $t_0, t_1, \ldots$ which are all given by the operational semantics and are such that $t = \biguplus t_i$.

**Proposition 8.8** Let $n \in \omega\Gamma c \in \exists_i w$ such that $fc \in w$ and $c$ is $\omega$-approximable. Let $d$ and $e$ be constraints that do not depend on hidden variables such that $(d \to e) \subseteq \exists_i f\Gamma$ where $(f, w) = E[A]\sigma_n s$. There is an $A(s)$-computation with a corresponding trace $t$ such that $\text{fn}\ t \supseteq (d \to e)$ and $\lim t = c$.

**Proof.** The proof is by induction on pairs $(n, A)$ under the lexical ordering.

Consider the agent $\bigwedge_{j \in I} A^j$. Let $(f_j, w_j) = E[A]\sigma_n (\pi_j s)\Gamma$ and let $q : \omega \to I$ be a mapping such that for each $j \in I \cap q^{-1}(i)$ is $j$ infinitely often. Let $c_0 = c$ and $c_{i+1} = f_q(i) c_i\Gamma$ for $i \in \omega$. Let $e' = \bigcup_{i \in \omega} d_i$. We have $e' \subseteq c\Gamma$ since $c$ is a fixpoint of all $f_j\Gamma$ for $j \in I$. On the other hand, $e'$ must be stronger or equal to $c\Gamma$ since $e' = \bigcup_{j \in I} f_j d_j = (f_j d_j) \subseteq e$.

To be able to apply the induction hypothesis we must show that $f_e c \in w_\Pi\Gamma$ for all $k \in I$. By Proposition 8.9 (below) we have $f_e = \bigcup_{j \in I} (\theta_k f_j)c$. Let $k \in I$ be fixed. We have $fc \in \theta_k w_k$. Thus $\Gamma fc = \theta_k c_k\Gamma$ for some $c_k \in w_k$. So $\theta_k^{-1}(fc) = c_k\Gamma$ and since $\theta_k^{-1}(fc) = \theta_k^{-1}(\bigcup_{j \in I} (\theta_j f_j)c) = f_k c$ we have $f_k c \in w_k$.

It is thus possible to apply the induction hypothesis and it gives us that we can for each $i \in \omega$ construct a $A'(\pi, s)$-computation (where $j = q(i)$) with trace $t_i\Gamma$ such that $\text{fn}\ t_i \supseteq d_i \to d_{i+1}$ and $\lim t_i = c$. By Theorem 6.11 there is an $A(s)$-computation with trace $t$ that satisfies the theorem.

Suppose $A = \exists_{i} A'$. Let $(f', w') = E[\exists_{i} A']\sigma_n s$. Let $d = \exists_i d$ and $e' = (f' \circ \exists_i d)$. Clearly $(d \to e') \subseteq f'$. Let $e' = \exists_{i} e$. To be able to apply the induction
hypothesis we must show that $f'c' \in w'$. First note that $\text{new}_{X}^{-1}(fc) \in w'$. We have

$$\text{new}_{X}^{-1}(fc) = \text{new}_{X}^{-1}((\text{new}_{X} f')c)$$

$$= \text{new}_{X}^{-1}((\text{new}_{X} \circ f' \circ \text{new}_{X}^{-1} \cup \text{id})c)$$

$$= (\text{new}_{X}^{-1} \circ \text{new}_{X} \circ f' \circ \text{new}_{X}^{-1} \circ \exists_{H} c) \cup (\text{new}_{X}^{-1} \circ \exists_{H})c$$

$$= f'(\exists_{X}c) \cup \exists_{X}c$$

$$= f'c'$$

By the induction hypothesis there is an $A'(s)$-computation with trace $t'$ such that $\lim t' = c'$ and $\text{fn}t' \sqsupseteq (d \rightarrow c')$. For $i \in r(t')$ let $d_i = \forall X(v(t') \Gamma) i.e.\Gamma$ the least $d_i$ such that $\exists X(d_i) \sqsupseteq v(t') \Gamma$ and $e_i = \exists X(v(t') \Gamma + 1)$. We can for each $i \in r(t')$ construct an $A(s)$-computation with trace $t_i$ such that $\text{fn}t_i \sqsupseteq (d_i \rightarrow e_i)$ and $\lim t_i = c'$. It follows from Theorem 6.11 that there is a computation $t$ with $\lim t = c$ and $\text{fn}t \sqsupseteq \bigcap_{i \in r(u)} \text{fn}t_i = E_X(\text{fn}u) \sqsupseteq (d \rightarrow c)$. □

The following proposition is used in the previous proof. It is difficult to give an intuitive explanation of what the proposition implies perhaps one can say that the proposition shows that by applying the appropriate projections to a set of closure operators we can make the closure operators independent with respect to hidden variables.

**Proposition 8.9** Let $\{f_j\}_{j \in I}$ be a family of closure operators. Let $f = \bigcap_{j \in I} \theta_j f_j \Gamma$ and $c$ be such that $\exists_H(fc) = c$. Let $d = \bigsqcup_{j \in I} (\theta_j f_j)c$. It follows that $d = fc$.

**Proof.** We begin by showing that $d \in f\Gamma i.e.\Gamma$ that $d$ is a fixpoint of $f$. To show that $d \in f\Gamma$ we must show that $d \in (\theta_j f_j)\Gamma$ for all $j \in I$. Let $j \in I$ be fixed.

$$(\theta_j f_j)d = (\theta_j f_j)(\bigsqcup_{k \in I} (\theta_k f_k)c)$$

$$= (\theta_j \circ f_j \circ \theta_k^{-1}c)(\bigsqcup_{k \in I} (\theta_k f_k \circ \theta_k^{-1}c))$$

$$= (\theta_j \circ f_j)(\bigsqcup_{k \in I} (\theta_k^{-1} \circ \theta_k f_k)c)$$

Now note that for $j = k$ we have $(\theta_j^{-1} \circ \theta_k \circ f_k)c = f_j c \Gamma$ and for $j \neq k$ we have $(\theta_j^{-1} \circ \theta_k f_k)c = \exists H(f_k c) \sqsupseteq \exists H(fc)$. Since $f_k c \sqsupseteq c = \exists H(fc) \sqsupseteq \exists H(fc) \sqsupseteq \exists H(\text{fn}c) \Gamma$ for $k \in I$ it follows that $(\bigsqcup_{k \in I} (\theta_k^{-1} \circ \theta_k f_k)c) = f_j c$. Thus

$$(\theta_j \circ f_j)(\bigsqcup_{k \in I} (\theta_k^{-1} \circ \theta_k f_k)c) = (\theta_j \circ f_j)(f_j c)$$

$$= (\theta_j \circ f_j)c$$

$$= (\theta_j f_j)c$$

It follows that $d \in (\theta_j f_j)\Gamma$ for all $j \in I \Gamma$ and thus $d \in f$.

Clearly $fc \sqsupseteq d \Gamma$ since $fc \sqsupseteq (\theta_j f_j)e \Gamma$ for all $j \in I$. We have $d \sqsupseteq c \Gamma$ since $(\theta_j f_j)c \sqsupseteq c \Gamma$ for $j \in I$. It follows that $fd \sqsupseteq fc \Gamma$ but since $d$ is a fixpoint of $f$ we have $d \sqsupseteq fc$. □

**Lemma 8.10** Let $t$ be a trace such that $\text{fn}t \sqsubseteq F(E[A][\sigma]s)$. There is an $A(s)$-computation with a corresponding trace $t'$ such that $t$ is a subtrace of $t'$.

**Proof.** For $i \in r(t)$ we have $(v(t)i \rightarrow v(t)i + 1) \sqsubseteq \text{fn}t$. For $i \in r(t)\Gamma$ let $t_i$ be a trace with $\lim t_i = \lim t$ and $\text{fn}t_i = (v(t)i \rightarrow v(t)i + 1)$. For a fixed $\Gamma$ there is an $n \in \omega$ such that $\text{fn}t_i \sqsubseteq F(E[I][\sigma]n, s)$. By Lemma 8.8 it follows that there is a computation
either a conjunction or a selection. If there is a computation with limit \( t \) and limit equal to that of \( t \), By Theorem 6.11 there is a computation \( t' \) such that \( \text{fn} t' \sqsupseteq \text{fn} t' \Gamma \) for each \( i \Gamma \) and thus \( \text{fn} t \sqsubseteq \text{fn} t' \). □

Intuitively, Tone would expect a correspondence between windows and the limits of fair computations. First we will consider the set of initially fair computations.

**Proposition 8.11** Suppose that \( A_0(s_0) : d_0 \rightarrow^* A(s) : d \) and that there is an \( \omega \)-approximable constraint \( e \in \mathcal{W}(E[A_0]s_0) \). There is an initially fair \( A(s) \)-computation with limit \( e \).

**Proof.** Note that \( \mathcal{W}(E[A_0]s_0) \cap \{ d_0 \}^n = \mathcal{W}(E[A]s) \cap \{ d \}^n \) if \( A_0(s_0) : d_0 \rightarrow^* A(s) : d \).

The proposition is proved by induction on pairs \((A, n)\) under the lexical ordering \( \Gamma \) where \( n \) is the number of computation steps from \( A_0(s_0) : d_0 \) to \( A(s) : d \). Let \( \{ e_i \}_{i \in \omega} \) be a chain in \( \mathcal{K}(\Omega) \) such that \( e = \bigcup_{i \in \omega} e_i \).

Suppose \( A \) is a conjunctive constraint \( c \). It follows that \( c \subseteq e \). The computation \((A(s) : e_0, A(s) : e_1, \ldots)\) is initially fair.

Suppose \( A \) is a conjunction. It follows from the computation rules that \( A_0 \) is either a conjunction or a selection. If \( A_0 \) is a selection we can find an agent \( A_1 \Gamma \) an oracle \( s \Gamma A_\Gamma \) constraint \( d_0 \) and \( n' < n \) such that \( A_0(s_0) : d_0 \rightarrow^* A_1(s_1) : d_1 \) and \( A_1(s_1) : d_1 \rightarrow^* A(s) : d \). It follows that we can apply the induction hypothesis (since \( (A, n') \prec (A, n) \) under the lexical ordering) and conclude that there is an initially fair \( A(s) \)-computation. Suppose \( A_0 = \bigwedge_{j \in I} A_0^j \) and \( A = \bigwedge_{j \in I} A^j \). We know that for all \( j \in I \) we have \( A_0^j(\pi_j s_0) : d \rightarrow^* A^j(\pi_j s) : d \). By the induction hypothesis it follows that there is an initially fair \( A^j(\pi_j s) \)-computation with limit \( e \) for each \( j \in I \). We can immediately form an initially fair \( A(s) \)-computation.

Suppose \( A \) is a selection \((c_1 \Rightarrow B_1 \left[ \ldots \right] c_m \Rightarrow B_m)\). If the oracle \( s \) begins with a 0 it follows that \( c_k \sqsubset e \Gamma \) for all \( k \leq m \). Thus the computation \((A(s) : e_0, A(s) : e_1, \ldots)\) is initially fair. If \( s \) begins with \( k \Gamma \) where \( 1 \leq k \leq m \Gamma \) it follows that \( c_k \subseteq e \). Let \( \Gamma \in \omega \) be such that \( e \sqsupseteq c_k \). We have an initially fair computation \((A(s) : e_0, A(s) : e_1, B_k(s') : e_1, B_k(s') : e_{k+1}, \ldots)\) where \( s' \) is the tail of \( s \).

Suppose \( A \) is an existential quantification \( \exists_X A' \). The case when \( A_0 \) is not an existential quantification can be treated using an argument similar to the case for conjunctions. Suppose \( A_0 \) is an existential quantification. Let \( e' = F(E[A]'s)(\exists_X e) \).

By Proposition 8.7 there is a trace \( t \) such that \( \lim t = e' \) and \( \text{fn} t \sqsubseteq (\exists_X (e) \rightarrow e') \) and \( \text{fn} t \sqsubseteq f \). By the induction hypothesis there is an initially fair \( A'(s) \)-computation with limit \( e' \). By Lemma 8.10 there is an \( A'(s) \)-computation with functionality stronger than \( \text{fn} t \) and limit \( e' \). By Theorem 6.11 it follows that these two computations may be combined into an initially fair \( A'(s) \)-computation with limit \( e' \) and functionality stronger than \( \text{fn} t \). Let \( B \) be the agent \( \bigwedge_{i \in \omega} \exists_X (e_i) \Gamma \) i.e., \( \Gamma \) a conjunction of agents in which each agent is a tell constraint \( \Gamma \) and let \( s' \) be an oracle such that \( \pi_0 s' = s \). Clearly there is an input-free \( A' \wedge B(s') \)-computation which is initially fair and has limit \( e' \). Let \( (w_i)_{i \in \omega} \) be the sequence of stores of the computation and \((A'_i)_{i \in \omega}\) the agents in the computation derived from the agent \( A' \Gamma \) the set of computation steps performed by \( A' \) in the computation.

We will now give \( c_i \) and \( d_i \Gamma \) for \( i \in \omega \Gamma \) such that \( \exists_X A'_i : d_i \) is an initially fair computation with limit \( e \).

Let \( c_0 = d_0 = \bot \). If \( i \in \Gamma \) let \( d_{i+1} = d_i \cup \exists_X (w_i) \) and \( c_{i+1} = w_i \). If \( i \not\in \Gamma \) let \( d_{i+1} = d_i \cup e_k \) if the \( k \)th step was performed by executing the \( k \)th factor of \( B \).
Let \( d_{i+1} = d_i \) otherwise. Let \( c_{i+1} = c_i \). It is straight-forward to verify that the constructed computation is indeed an initially fair computation with limit \( e \).

Suppose \( A \) is a call \( p(X) \). If the definition of \( p \) is of the form \( p(Y) :: A' \) it follows from the computation rules that

\[
(A(s) : c_0, \{Y \rightarrow X\} A'(s) : c_0, \{Y \rightarrow X\} A'(s) : c_1, \ldots)
\]

is an initially fair computation.

\( \square \)

As we have shown the existence of initially fair computations for a given member of a window \( \Gamma \) we have actually done most work necessary to prove the existence of fair computations. For a given member \( c \) of a window \( \Gamma \) we need to construct a corresponding fair computation which has \( c \) as limit.

**Lemma 8.12** Let \( c \) be an \( \omega \)-approximable constraint in \( W(E[A] \sigma s) \). There is a fair \( A(s) \)-computation with limit \( c \).

**Proof.** We will construct a family of initially fair chains \((L^k_i)_{i \in \omega}\) such that \( L^0_i = [A(s) : d] \Gamma \) for some \( d \Gamma \) and each chain has limit \( c \). Further, we make sure that \( L^0_i \xrightarrow{\delta} L^{i+1}_i \) and for each \( k \) and \( i \) there is a \( k' \) such that \( L^k_i \xrightarrow{\delta} L^{k'}_0 \). We will show that given this the sequence \((L^k_i)_{i \in \omega}\) is a fair chain with limit \( c \).

Let \((L^0_i)_{i \in \omega}\) be an initially fair \( A(s) \)-computation with limit \( c \). For \( k \geq 0 \) let \( L^{k+1}_i = L^0_i * L^1_i * \cdots * L^i_{k-1} * L^i_k \) and let \( \{L^k_i\}_{i \geq 0} \) be such that \((L^k_i)_{i \in \omega}\) is initially fair.

It is easy to verify that the family \((L^k_i)_{i \in \omega}\) satisfies the properties mentioned above. It follows immediately that \((L^k_i)_{i \in \omega}\) is a fair chain. To verify that \((L^k_i)_{i \in \omega}\) is fair, consider a suffix \((L^k_i)_{i \geq m}\). Since \((L^m_i)_{i \in \omega}\) is initially fair and because of Lemma 6.8, we find that the suffix must be initially fair. It follows that \((L^k_i)_{i \in \omega}\) is a fair chain.

We are now ready to give the proof of Theorem 8.5. Recall that \( t \) is a trace such that \( \text{lim} \ t \in W(E[A] \sigma s) \) and \( \text{fn} t \in F(E[A] \sigma s) \) and we want to show the existence of a trace \( t' \in \mathcal{O}_H[A] \) such that \( t \) is a subtrace of \( t' \).

**Proof.** (Theorem 8.5) By Lemma 8.12 there is a fair \( A(s) \)-computation with limit \( t \). By Lemma 8.10 there is an \( A(s) \)-computation with functionality greater than or equal to \( \text{fn} t \) and limit equal to \( t \). By Theorem 6.11 the two computations can be combined into a fair computation with functionality at least as strong as \( \text{fn} t \) and limit equal to \( t \). \( \square \)

9 Category-theoretic semantics

In this section we will use a powerdomain construction by Lehmann [910] to devise a fixpoint semantics which is more abstract than the oracle-based fixpoint semantics. This powerdomain construction has previously been used by Abramsky [11] Panangaden and Rutten [17] and Nyström and Jonsson [15] to give the fixpoint semantics of various forms of nondeterministic programming languages.

Lehmann’s construction relies on a special type of categories called \( \omega \)-categories.

**Definition 9.1** An \( \omega \)-category is a category which has an initial object and in which all \( \omega \)-chains have colimits.

An \( \omega \)-functor is a functor which preserves colimits of \( \omega \)-chains. \( \square \)
It is easy to see that a cpo or a complete lattice can also be seen as a $\omega$-category, and that a continuous function over a cpo or complete lattice is an $\omega$-functor over the corresponding category.

The following construction, which is by Lehmann, gives a powerdomain for a given cpo $(D, \sqsubseteq)$.

**Definition 9.2** Assuming a cpo $(D, \sqsubseteq)$ the objects and arrows in the corresponding powerdomain $CP(D)$ are as follows. The objects are multisets over $D$ which we represent as sets of pairs $x_\gamma$ where $x \in D$ and $\gamma$ is some tag. We will not make any assumptions about the tags other than that they exist in sufficient numbers to represent the multisets we are interested in. The tags will often be omitted when they are clear from the context.

An arrow $r : A \to B$ of $CP(D)$ is a relation $r \subseteq A \times B$ such that for each $y \in B$ there is a unique $x \in A$ such that $\langle x, y \rangle \in r$ and whenever $\langle x, y \rangle \in r$ we have $x \sqsubseteq y$. We can view the arrow $r$ as representing a function $r^{-1} : B \to A \Gamma$ satisfying $r^{-1}y \sqsubseteq y\Gamma$ for any $y \in B$.

For a diagram $A_0 \xrightarrow{r_0} A_1 \xrightarrow{r_1} \ldots$ the colimit has the following form. Let

$$S = \{ (x_i)_{i \in \omega} \mid x_{i+1} = r_i^{-1}x_i \text{ for } i \in \omega \}.$$

The colimiting object is now $B = \{ \sqcup_{i \in \omega} x_i \mid (x_i)_{i \in \omega} \in S \}$ together with the arrows $f_i : A_i \to B$ such that $f_i^{-1}(\sqcup_{i \in \omega} (x_i)) = x_i\Gamma$ for $\sqcup_{i \in \omega} (x_i) \in S$.

### 9.1 Constructions

Given objects $A$ and $B$ of a cpo the *product* $A \times B$ is (provided that it exists) an object $C$ together with arrows $r_1 : C \to A$ and $r_2 : C \to B\Gamma$ which satisfies the following properties. For any object $X$ with arrows $f_1 : X \to A$ and $f_2 : X \to B$ there is a unique arrow $g : X \to C$ such that $f_1 = g \circ r_1$ and $f_2 = g \circ r_2$.

In the category $CP(D)$ the product is simply the disjoint union. Given objects $A$ and $B\Gamma$ let $C = \{ x_\gamma \mid x_\gamma \in A \} \cup \{ y_\gamma \mid y_\gamma \in B \}$. Let $r_1^{-1}(x_\gamma) = x_{(\gamma, 1)}\Gamma$ for $x_\gamma \in A\Gamma$ and similarly $r_2^{-1}(y_\gamma) = y_{(\gamma, 2)}\Gamma$ for $y_\gamma \in B$. It is easy to check that this is in fact the product. It is a theorem of category-theory that $\cdot$ is an $\omega$-functor on both arguments when defined on all pairs of objects.

The product will be used to model the non-deterministic choice between two alternatives. We will write $\Downarrow$ for the product.

The dual notion of product coproduct will also be used in the category-theoretic fixpoint semantics. Given objects $A$ and $B$ of a cpo the coproduct $A + B$ is (provided that it exists) an object $C$ together with arrows $r_1 : A \to C$ and $r_2 : B \to C\Gamma$ which satisfies the following properties. For any object $X$ with arrows $f_1 : A \to X$ and $f_2 : B \to X$ there is a unique arrow $g : C \to X$ such that $f_1 = r_1 \circ g$ and $f_2 = r_2 \circ g$.

If $D$ is a lattice and $A$ and $B$ are objects of $CP(D)$ the coproduct $C = A + B$ can be formed by

$$C = \{ z_{(\gamma_1, \gamma_2)} \mid z = x \sqcup y, x_{\gamma_1} \in A, y_{\gamma_2} \in B \}.$$

The arrow $r_1 : A \to C$ is given by $r_1^{-1}z_{(\gamma_1, \gamma_2)} = x_{\gamma_1}\Gamma$ for $x_{\gamma_1} \in A\Gamma y_{\gamma_2} \in B$ and $z = x \sqcup y$. The arrow $r_2 : B \to C$ is similar.
The definition of coproduct can easily be generalised to arbitrary sets of objects. In this case we will use the symbol $\sum$ for the coproduct.

Given categories $\mathbf{A}$ and $\mathbf{B}$, the product $\mathbf{A} \times \mathbf{B}$ is the category where the objects consist of pairs of one object from $\mathbf{A}$ and one object from $\mathbf{B}$, and the arrows are pairs of arrows from $\mathbf{A}$ and $\mathbf{B}$ such that $\langle f, g \rangle : \langle A_1, B_1 \rangle \to \langle A_2, B_2 \rangle$ is an arrow of the product category if $f : A_1 \to A_2$ is an arrow of $\mathbf{A}$ and $g : B_1 \to B_2$ is an arrow of $\mathbf{B}$. For an ordered finite set $S$ with $n$ elements we write $A^S$ for the category $\mathbf{A}_1 \times \ldots \times \mathbf{A}_n$. If $a \in S$ is the $k$th element of $S$ let $\text{index}_a$ be a functor $\text{index}_a : A^S \to \mathbf{A}$ such that for objects $\langle A_1, \ldots, A_n \rangle = A_k$ and for arrows $\langle f_1, \ldots, f_n \rangle = f_k$.

We will also need the following result regarding the construction of $\omega$-functors $\Gamma$ which is by Lehmann [10]. Given a continuous function $f : D_1 \to D_2$ define the operation $\hat{f}$ as follows. For an object $A \in D_1 \Gamma$ let

$$\hat{f}(A) = \{y_i \mid y = f(x), x_i \in A\},$$

and for an arrow $r : A_1 \to A_2$ in $CP(D_1)$ we take $\hat{f}(r) : \hat{f}(A_1) \to \hat{f}(A_2)$ to be given by

$$\hat{f}(r) = \{\langle x_i, y_i \rangle \mid \langle x_i, y_i \rangle \in r, y = f(x), y' = f(x')\}.$$

**Proposition 9.3** Let $f : D_1 \to D_2$ be a continuous function. Then $\hat{f} : CP(D_1) \to CP(D_2)$ is an $\omega$-functor.

### 9.2 The Powerdomain of Trace Bundles

In this section we will consider a fixpoint semantics based on the powerdomain of trace bundles. Let $\text{Proc} = CP(\text{BUNDLE})$.

#### 9.2.1 Basic operations

**Tell constraints** First to give the semantics of a tell constraint $c$ we use the following constant functor which returns a singleton set consisting of a trace bundle with a functionality which adds $c$ to the store and a window that makes sure that $c$ is in the store. Let

$$\langle c \rangle = \{ \langle \bot \to c, \{c\}^\omega \rangle \}.$$

**Disjoint union** One operation that comes with the categorical powerdomain is the disjoint union $\parallel$. The disjoint union is a functor of arbitrary arity over the processes. This operation will be used we give the semantics of non-deterministic choice.

Given $\omega$-functors $F_1, F_2 : \text{Env} \to \text{Proc}$ we can construct an $\omega$-functor $\parallel(F_1, F_2) : \text{Env} \to \text{Proc}$ that returns the disjoint union of the results of applying $F_1$ and $F_2$ to the argument. We will take advantage of this to simplify the presentation of the categorical semantics $\Gamma$ and not distinguish explicitly between $\parallel$ as a functor over processes and a functor over functors from environments to processes.
Parallel Composition  A rather appealing property of the categorical semantics is the similarity between coproduct and parallel composition. For processes $P_0, P_1, \ldots$ the coproduct can be formed by

$$\sum_i P_i = \{ \langle f_i, w_i \rangle \mid \langle f_i, w_i \rangle \in P_i, \text{ for all } i \}$$

The coproduct corresponds to a parallel composition where the processes do not have a private state since different processes may refer to the same hidden variable. To obtain the normal parallel composition of processes we must first apply the projection operators in the same way as in the oracle semantics. In other words the parallel composition of a family $\{P_j\}_{j \in I}$ of processes is given by the following expression

$$\sum_{j \in I} \theta_j P_j,$$

where renamings have been extended to processes by the definition

$$\theta P = \{ \langle \theta f, \theta w \rangle \mid \langle f, w \rangle \in P \}.$$

To simplify the presentation we will also use $\sum$ as a higher-order functor that takes a family of functors $\{F_j : \text{Env} \rightarrow \text{Proc}\}_{j \in I}$ and returns a new functor $\sum_{j \in I} : \text{Env} \rightarrow \text{Proc}$ defined according to the equation $(\sum_{j \in I} F_j)A = \sum_{j \in I} F_j A$.

Ask Constraints  Ask constraints are modelled using a functor select($c$) : Proc $\rightarrow$ Proc which for a given constraint $c$ takes a process and returns a process consisting of trace bundles which do not generate any output until $c$ is satisfied and which require that $c$ is eventually satisfied.

$$\text{select}(c)P = \{ \langle c \rightarrow f, \{c\}^u \cap w \rangle \mid \langle f, w \rangle \in P \}$$

Unless  Given constraints $c_1, \ldots, c_n$ the constant functor unless($c_1, \ldots, c_n$) is defined. It returns a singleton set containing the trace bundle which is always passive and requires that no $c_k$ is ever satisfied.

$$\text{unless}(c_1, \ldots, c_n) = \{ \langle \text{id}, \emptyset \mid \{c_1\}^u \cup \ldots \cup \{c_n\}^u \rangle \}$$

We will use this functor to model the case when in a selection no alternative is ever chosen.

Existential quantification  Existential quantification is treated as in the oracle semantics; the new$_X$ renaming is applied to change the name of the (visible) variable $X$ into a hidden one. Extend new$_X$ to be a functor over the CP(Bundle)Fi.e.Plet

$$\text{new}_X P = \{ \langle \text{new}_X f, \text{new}_X w \rangle \mid \langle f, w \rangle \in P \}.$$

9.2.2 The categorical fixpoint semantics  The categorical fixpoint semantics is given in Figure 4.
For each agent $A$ define a functor $\mathcal{E}[A] : \text{Proc}^N \to \text{Proc}$ according to the following equations

$$
\begin{align*}
\mathcal{E}[c] &= \langle c \rangle \\
\mathcal{E}[\bigwedge_{j \in I} A_j] &= \sum_{j \in I} (\theta_j \circ \mathcal{E}[A_j]) \\
\mathcal{E}[\exists x A] &= (\bigcup_{0 < k \leq n} \text{select}(c_k) \circ \mathcal{E}[A_k]) \\
&\quad \text{unless}(c_1, \ldots, c_n) \\
\mathcal{E}[\exists X A] &= \text{new}_X \circ \mathcal{E}[A] \\
\mathcal{E}[P(X)] &= \{ \Delta \to X \} \circ \text{index}_P
\end{align*}
$$

For a program $\Pi$ define a functor $\mathcal{P}[\Pi] : \text{Proc}^N \to \text{Proc}^N$ according to the equation

$$
\mathcal{P}[\Pi] = \langle \{ Y \to \Delta \} \circ \mathcal{E}[A] \rangle_{P \in N},
$$

where for each $p \in N$ the definition in $\Pi$ is assumed to be of the form $p(Y) :: A$ for some variable $Y$ and some agent $A$.

Figure 4: The categorical fixpoint semantics

10 Comparison between the oracle semantics and the categorical semantics

We want to show that the categorical fixpoint semantics gives the same set of traces as the oracle-based semantics.

10.1 Augmenting the oracle semantics

Recall that the semantics domain for agents in the oracle semantics $\Gamma A$ is the set of functions from oracles to trace bundles that is pairs consisting of of a closure operator $f$ and a window $w$. We will give an abstraction operator that maps each element of $A$ to an object in the category of processes.

It is easy to see that each infinite oracle gives a (possibly empty) set of traces $\Gamma$ so we might define an abstraction operator that maps each element $a : \text{ORACLE} \to \text{BUNDLE}$ to a multiset

$$
\{ a(s) \mid s \text{ is an infinite oracle} \}
$$

However this construction is problematic since the minimal member of $A\Gamma$ which is $\lambda a : \langle \text{id}, U \rangle \Gamma$ would be mapped to a process object where each trace bundle had an infinite multiplicity $\Gamma$ and not to the initial object of the category of processes. Also $\Gamma$ the introduction of such multiplicities of elements appears to be rather unnatural.

For a given agent $\Gamma$ we must find a set of oracles that is sufficient to generate trace bundles corresponding to all possible computation paths $\Gamma$ but which still bears some relationship to the choices performed by the agent. One possible way to determine the set of oracles to is to examine the result of the oracle-semantics. Suppose $a : \text{ORACLE} \to \text{BUNDLE}$. Let $S$ be the smallest set (with respect to the inclusion order) such that $a(s') = a(s)$ for $s \in S$ and $s' \geq s$. However $\Gamma$ it is unclear whether this construction is continuous. Also $\Gamma$ there does not seem to be any relationship
between the multiplicity of the result of the category-theoretic semantics and the set of oracles.

Instead we define a semantic function \( D \) which will provide us \( \Gamma \) for each agent \( \Gamma \) with a set of oracles which are sufficient to determine the set of traces generated.

For a partial order \( P \) say that a set \( S \subseteq P \) is anti-consistent if no two elements in \( S \) have an upper bound in \( P \) i.e. for all \( x, y \in S \) such that \( x, y \leq z \Gamma \) for some \( z \in P \) we have \( x = y \).

Let \( AC \) be the set of anti-consistent subsets of oracle. For \( a, b \in AC \) say that \( a \subseteq b \) if \( a^u \supseteq b \) in oracle. It turns out that \( AC \) forms a cpo under this ordering with \( \{ e \} \) and least element. Now we can define \( D[A] : AC^N \to AC \) inductively for each agent \( A \).

\[
D[c] \delta = \{ e \} \\
D[\bigwedge_{j \in I} A_j] \delta = \{ s_j \in D[A_j] \delta, \text{ for } j \in I \} \\
D[(c_1 \Rightarrow A_1 \mid \ldots \mid c_n \Rightarrow A_n)] \delta = \{ k.s \mid 1 \leq k \leq n, s \in D[A_k] \delta \} \\
\cup \{ 0 \} \\
D[3_X A] \delta = D[A] \delta \\
D[P(X)] \delta = \delta P
\]

For a program \( \Pi \) we can now define a functor \( Q[\Pi] : AC^N \to AC^N \) according to the equation

\[
Q[\Pi] \delta P = D[\Pi] \delta,
\]

where the definition of \( P \) in \( \Pi \) has the form \( P(X) :: A \). It should be clear that for a program \( \Pi \) and agent \( \Lambda \Gamma \) and \( \delta \) the least fixpoint of \( Q[\Pi] \Gamma \) the set \( S = D[A] \delta \) is sufficient to determine the set of traces generated by the oracle semantics. For example for the agent

\[
A = (X = 1 \Rightarrow Z = 3 \mid Y = 2 \Rightarrow W = 5)
\]

the set of trace provided is \( D[A] \delta = \{ 0, 1, 2 \} \).

### 10.2 An intermediary category

To facilitate the comparison between the oracle semantics and the categorical fixpoint semantics we will give a semantics which between the two semantics.

Recall that the objects of \( CP(BUNDLE) \) are multisets of trace bundles where the elements of multisets are tagged with some arbitrary value to distinguish between multiple occurrences of an element. An arrow \( P_1 \rightarrow P_1 \) of \( CP(BUNDLE) \) is a reverse mapping \( r^{-1} : P_2 \rightarrow P_1 \) mapping each tagged element of \( P_2 \) to an element of \( P_1 \) which is smaller or equal.

**Definition 10.1** Let \( \text{INTER} \) be the sub-category of \( CP(BUNDLE) \) where the objects and arrows satisfy these additional requirements.

1. The objects are multi-sets of trace bundles where the tags are drawn from the set of oracle and member of an object has a unique tag and the tags of members of an object form an anti-consistent set.
2. The arrows \( r : P_1 \to P_2 \) satisfy the following for all \( \langle f_1, w_1 \rangle_{s_1} \in P_1 \) and \( \langle f_2, w_2 \rangle_{s_2} \in P_2 \). It holds that \( r^{-1}(\langle f_2, w_2 \rangle_{s_2}) = \langle f_1, w_1 \rangle_{s_1} \) exactly when \( s_1 \subseteq s_2 \).

For an oracle \( s \) and an object \( P \) of \textsc{Inter} we will write \( s \in P \) to indicate that there is a trace bundle in \( P \) with tag \( s \) and \( P(s) \) for that particular trace bundle.

Obviously \textsc{Inter} is a sub-category of \textsc{CP(Bundle)}. It should also be clear that that \textsc{Inter} has an initial element given by \( \langle \text{id}, \mathcal{U} \rangle \), which is also an initial element of \textsc{CP(Bundle)}. For an \( \omega \)-chain

\[
P_0 \xrightarrow{\rho_0} P_1 \xrightarrow{\rho_1} P_2 \xrightarrow{\rho_2} \ldots
\]

the colimit can be given by

\[
P = \{ \langle \bigcup_{i \in \omega} P_i(s_i) \rangle_s \mid s_i \in P_i, \text{ for } i \in \omega \text{ and } s = \bigcup_{i \in \omega} s_i \}.
\]

Clearly \( \omega \)-colimit in the category \textsc{Inter} coincides with the corresponding colimit in \textsc{CP(Bundle)}.

So \textsc{Inter} is a sub-\( \omega \)-category of \textsc{CP(Bundle)}.

### 10.3 Refining the basic operations

The basic operations of the categorical powerdomain was given without regard to the choice of tags of the members of the multisets. This was of course the natural way to define things but we shall see that by strengthening the definitions of the basic operations of the categorical fixpoint semantics it is actually possible to give the categorical fixpoint semantics in the intermediate category.

The idea is that we refine the basic operations given for the categorical semantics to make sure that all operations are well-defined in the intermediate category.

**Tell constraints** The semantics of the tell constraint is given with a constant functor which returns a singleton multiset. We just need to give the single member of the multiset a tag which is an oracle. Let

\[
\langle c \rangle = \{ \langle \bot \to c, \{c\}^n \rangle \}.
\]

**Disjoint union** In the categorical powerdomain disjoint union correspond to category-theoretic product. The product in the intermediate category is not a disjoint union due to the restrictions on arrows but it is still possible to define a functor which returns a disjoint of multi-sets.

For a family \( \{ P_k \}_{0 \leq k \leq n} \) of multi-sets let

\[
\biguplus_{0 \leq k \leq n} P_k = \{ \langle f, w \rangle_{k, s} \mid \langle f, w \rangle_{s} \in P_k, 0 \leq k \leq n \}.
\]

It is easy to see that the operation is indeed a functor in the intermediate category and that it is a refinement of the disjoint union of the category-theoretic powerdomain.
Parallel Composition  Coproduct in the intermediate category corresponds to coproduct in the $CP$(bundle).

$$\sum_{j \in I} P_j = \{\langle \cap_{j \in I} f_j, \cap_{j \in I} w_j \rangle_s | \langle f_j, w_j \rangle_s \in P_j, \text{ for } j \in I, \text{ and } s = \cap_{j \in I} s_j \}.$$  

Selections  It is straightforward to refine the functor $\text{select}(c) : \text{Proc} \rightarrow \text{Proc}$ to a functor over the intermediate category. Let

$$\text{select}(c)P = \{ \langle c \rightarrow f, \{c\}^u \cap w \rangle_s | \langle f, w \rangle_s \in P \}.$$  

The constant functor $\text{unless}(c_1, \ldots, c_n)$ is treated in the same way as the constant functor $\langle c \rangle$ which gives the semantics for tell constraints. Let the single element in the multiset returned by $\text{unless}(c_1, \ldots, c_n)$ be tagged by the oracle $\epsilon$.

Existential quantification  In the categorical semantics, existential quantification is obtained by applying the function $\text{new}_X$ to each trace bundle. Refining this operation to the intermediate category is done by retaining the tags of the argument to the functor $\text{Term}$ $\text{new}_X P = \{ \langle \text{new}_X f, \text{new}_X w \rangle_s | \langle f, w \rangle_s \in P \}$.

10.3.1 Intermediate fixpoint semantics  The semantic equations for the intermediate fixpoint semantics are the same as for the categorical powerdomain semantics.

10.3.2 Relation with the oracle semantics  For a given program and agent $\Gamma$, the oracle semantics gives a function $a$ which maps oracles to trace bundles. In some cases the resulting trace bundle corresponds to an empty set of traces and in other cases the trace bundle was provided by applying the function $a$ (the ‘semantics’ of the agent) to a weaker oracle. Augmenting the oracle semantics with the semantic function $D$ and $Q$ provides us for each agent $\Gamma$ with a set of oracles which is sufficient to generate all traces of that agent. So if the oracle semantics of an agent is $a\Gamma$ and the corresponding set of oracles is $S\Gamma$ we can give the set of trace bundles as the set $\{as | s \in S\}$. The corresponding mapping to the intermediate category is

$$a\Gamma(a, S) = \{\langle as \rangle_s | s \in S \}.$$  

11 Conclusions

This paper presents two fixpoint semantics for concurrent constraint programming. Both semantics take into account infinite computations and fairness between processes.  

First an operational semantics is given which includes the concept of oracles. The idea is that the non-deterministic choices of a process are thought of as being governed by an oracle and thus a process with a given oracle is essentially deterministic.  

Before the fixpoint semantics is presented a number of confluence properties are shown to hold. In short the result of the confluence properties is that a countable set of computations with a given agent and oracle can be combined to yield a stronger computations. If one of the computations in the set is fair then it is possible to find a computation which is fair and stronger than all computations in the set.
The first fixpoint semantics is based on oracles. The idea is that the semantics of an agent is given with respect to a particular oracle. It turns out that even though non-determinism is effectively eliminated by the use of oracles it is still not possible to give a fully abstract fixpoint semantics for agent-oracle pairs. (This negative result is rather surprising and may be of interest in itself.)

To get a fixpoint semantics it is necessary to make the local state part of the semantics. This is done by introducing a class of variables called the hidden variables and two types of renaming operations.

The oracle fixpoint semantics of an agent together with an oracle consists of two parts: the functionality which describes the input-output relation of the agent and the window which gives conditions in the input under which the particular combination of agent and oracle may produce a computation.

The oracle fixpoint semantics is rather straightforward (even though its proof of correctness is not) and it is easy to see exactly which aspects of the semantics that makes it less than fully abstract (that is, the use of oracles and the visibility of the local state.)

The oracle semantics has some advantages compared to other approaches to the semantics of non-deterministic programming languages which might make it useful in the analysis of concurrent programs. Other approaches, i.e., giving the semantics as a set of traces or modeling concurrency as interleaving in an operational semantics, give a set of possible branches which is exponential in the length of the computation. In contrast, the oracle fixpoint semantics gives a set of possible branches which is exponential in the number of actual non-deterministic choices. While this is still of high complexity, it is nevertheless a significant improvement.

The second fixpoint semantics is the categorical semantics. The oracles give us a tree of alternative branches and the use of Lehmann’s categorical powerdomain allows us to put the branching information in the arrows of the category that is the Lehmann powerdomain. The resulting fixpoint semantics is a bit simpler than the oracle semantics and one might argue also a bit more abstract.

References


