UPPSALA THESES IN COMPUTING SCIENCE
24

Denotational Semantics for Asynchronous Concurrent Languages

SVEN-OLOF NYSTRÖM

Computing Science Department
Uppsala University

UPPSALA 1996

(Dissertation for the Degree of Doctor of Philosophy in Computing Science presented at Uppsala University in 1996)

Abstract

Nyström, S. 1996: Denotational Semantics for Asynchronous Concurrent Languages. *Uppsala Theses in Computing Science* 24. 182 pp. Uppsala. ISSN 0283-359X, ISBN 91-506-1154-2.

Asynchronous concurrent languages are programming languages that allow a problem to be solved by a set of cooperating processes, and where the processes communicate by asynchronous message passing, that is, a message can be sent regardless of whether the receiver is ready to accept the message or not.

The focus is on a model of programming called *concurrent constraint programming* (ccp), which it will be argued consists of the elements essential to concurrent programming. An important goal is to find semantic models that focus on the *external* behaviour of programs, and where the interaction between a program and its environment is modelled as abstract dependencies between input and output. Throughout the thesis it is assumed that a semantics for a concurrent language should consider not only the results of finite, terminating computations, but also the results of infinite computations.

We give a fully abstract semantics for ccp, which is however not a fixpoint semantics. We give two proofs of full abstraction; one which depends on the use of an infinite conjunction of processes, and one which makes some assumptions about the constraint system but only requires finite conjunctions.

We show that for a large class of concurrent programming languages there is no denotational semantics which is a fixpoint semantics and fully abstract. Similar results have been presented by other authors, but the result presented here is more general.

We introduce an operational semantics of ccp based on the use of *oracles*. An oracle describes the sequence of non-deterministic choices to be made by a process. We show a confluence property for ccp which concerns infinite sets of computations and infinite sequences of computation steps. As far as I know, no similar confluence property has been described for ccp or any other concurrent programming language.

We give a fixpoint semantics for ccp, based on the oracle semantics. In this semantics, the oracles are explicit. By abstracting the oracles from the fixpoint semantics we obtain a slightly more abstract version of the fixpoint semantics in which the semantic domain is a category.

Sven-Olof Nyström, Computing Science Department, Uppsala University, Box 311, 751 05 Uppsala, Sweden. Phone: 018-18 10 57. Fax: 018-51 19 25.

ISSN 0283-359X
ISBN 91-506-1154-2

Contents

Chapter 1

Introduction

This thesis is about the semantics of concurrent programs. Below I state the underlying notions which led to the results presented in the thesis.

I choose to focus on a model of programming called *concurrent constraint programming* (ccp), which I will argue consists of the elements essential to concurrent programming.

Throughout the thesis I assume that a semantics for a concurrent language should consider not only the results of finite, terminating computations, but also the results of infinite computations.

The semantic models I develop in the thesis are intended to describe the *external* behaviour of programs.

1.1 Semantic models

In the context of computer science, *semantics* is the meaning of programs. The semantics of a programming language can often be stated in words, but experience has shown that this often leads to ambiguities.

One rather straight-forward way to state the semantics of a programming language in a formal way is to use an *operational semantics*. Here, the execution of a program is modelled in a stepwise manner. Using an operational semantics we can, starting with a program and an input, work out what the program should compute, but this does not give us a general understanding of what the program *means*. Also, an operational semantics makes assumptions about how a program is executed, while there may be many different ways to implement a programming language.

Another way to state the semantics of programs is to use a *denotational semantics*. Here, the meaning of programs and components of programs are given as elements in a mathematical structure. A denotational semantics is *compositional*, meaning that the semantics of a programming construct can be determined from the semantics of its components. For instance, it should be possible to determine the semantics of a while-loop (containing a boolean test and a body) given the semantics of the test and of the body of the loop. If we replace the body of the loop with another body that has the same semantics, we expect the semantics of the loop to be the same. Similarly, replacing one sorting routine with another routine that has the same semantics should not change the semantics of the program.

One important aspect of denotational semantics is the use of *fixpoints*. When giving the semantics of a procedure that is defined in terms of itself, i.e., it is recursive, it is natural to see the procedure definition as an equation to be solved. In this case, the solution we are interested is the least specific solution. In the traditional denotational semantic models, the least specific solution is the least fixpoint of a function which has been obtained from the recursive definition.

We do not want a denotational semantics to contain any redundant information; ideally, a denotational semantics should give exactly the information needed to determine if two programs (or program fragments) are equivalent. When this holds, we say that semantics is *fully abstract*. When we have a fully abstract semantics, and we compare the denotational semantics of two programs (or program fragments), and we discover that the two programs have different denotational semantics, this might either be because the two programs really behave differently, or that when we make the programs part of a larger program there may be a detectable difference in behaviour. As long as we always can find a reason for programs having different denotational semantics, we say that the denotational semantics is fully abstract.

1.2 Concurrency

In many programming problems, it is very natural to model a program as a set of cooperating processes. The processes are up to a point independent of each other, but exchange information. A programming language is concurrent if it is possible to write programs which consist of communicating processes. It should be stressed that concurrency is about expressiveness; even though in theory all programming problems can be solved in a sequential programming language, some programming problems are much easier to solve in a concurrent language.

Typical programming problems which are natural to solve using concurrent programming include programs that are distributed on a network of computers, and programs which communicate with many different external units, consider for example a booking system which interacts with a large number of users.

Now, if we want a concurrent programming language, what basic operations are necessary for concurrent programming? It is easy to see that two things are essential. First, there must be a way to create processes. Second, there must be a way for processes to communicate. As a third point we could mention synchronisation between processes. However, we consider communication to implicitly involve an element of synchronisation, in that a process may wait for a message from another process.

Note that the result of a concurrent computation is often not completely determined by the input to the program. When this is the case, we say that the computation is *non-deterministic*. Non-determinism occurs for example when two processes send messages to a third process. The behaviour of the third process depends on in which order it receives the messages. That a computation is non-deterministic is problematic, both from a practical and a theoretical point of view. Non-determinism complicates the testing of programs, since we cannot be sure that the repeated execution of the same program will result in the same behaviour. Also, it is quite possible that different implementations may favour different non-deterministic choices, so that moving a program from one computer to another results in a different behaviour. It is also well-known that non-determinism in a programming language makes it harder to give a denotational semantics for the language. Regardless of these problems, there is no way to design a concurrent language without non-determinism that will not restrict the expressiveness of the language.

1.3 Concurrent constraint programming

In my opinion, concurrent constraint programming (ccp) constitutes a very natural model of concurrent programming. Any programming model must provide mechanisms to allow the program to receive input, to allow an intermediate result to be communicated between different parts of the program, and to allow the program to present the computed result. In a concurrent language there must also be a mechanism for communication between processes.

In ccp, all forms of communication are served by a single mechanism, the store. The store is simply a collection of facts gathered during the computation. (The facts are called *constraints*, for historical reasons.) There are two types of operations on the store, *tell* which add a constraint to the store, and *ask*, which succeed when a given constraint is entailed by the store.

An important feature of ccp is that the store grows monotonically, in that constraints may be added to the store, but are never removed. It follows that an ask constraint that succeeds at one point in time will also succeed if tested later. This makes it easier to reason about ccp, formally and informally, and is also helpful when implementing ccp on a computer with distributed memory.

1.4 Why ccp?

When one discusses the semantics of programming languages, it helps to have a language which has a simple formal definition, since we want to reason about the language formally. On the other hand, we do not want the language to be so restrictive that we cannot reason about common programming techniques.

It seems to me that ccp fits these requirements quite well. The use of the store as a medium for communication gives a communication mechanism which is very simple, but is still quite general and powerful. Given the assumption that the store should be the medium for communication, the other aspects of concurrent constraint programming follow quite naturally. It is difficult to imagine that a concurrent language with non-determinism, recursion and data hiding could be simpler than ccp.

1.5 Why infinite computations?

In the traditional formal models of sequential computation, a program begins by reading its input, then computes, and then terminates, at which point it presents the computed result. In other words, a sequential program that does not terminate has not made its results available to the outside, so we can say that it has not really produced anything. The situation in concurrent programming languages is different. A process can read input and produce output without terminating. In fact, one can argue that the only difference between a process that terminates and a process that enters an infinite loop and does no more communication is that the terminating process is more efficient. After all, the output from the two processes is the same.

There are many completely reasonable programs that are written so that they could run indefinitely, if we ignore the physical limitations. Consider, for example, an operating system, or the software of a telephone exchange. In all these cases, there is nothing inherent in the programs that would prevent them from executing indefinitely, if we had completely reliable hardware and infinite patience. Even a simple interactive program like a word processor could run indefinitely, if the user keeps editing, and editing, and editing, ...

Of course, the fact that the behaviour of programs is well-defined even in the case of infinite executions does not necessarily mean that we need to make infinite observations. Since the observations we can make about a running program are limited to finite prefixes of the computation, it may be that finite observations are sufficient to determine the behaviour of a program performing an infinite computation. If we consider *deterministic* programs, it is indeed the case that the infinite behaviour of a program is completely determined by the finite observations we can make. For example, if a deterministic program generates the output

$$a_0, a_1, a_2, \ldots,$$

it is sufficient that we can make the observation $a_0, a_1, \ldots, a_n$, for any $n \geq 0$, to allow us to uniquely determine the complete result of the computation.

In contrast, for non-deterministic programs finite observations are not sufficient to determine the infinite behaviour. Worse, finite observations are not even sufficient to distinguish between programs that always terminate and programs that may not terminate. We will give a couple examples to illustrate the problems with only allowing finite observations.

For our examples we introduce a simple non-deterministic programming language, where the syntax of statements is as follows.

$$S ::= \text{print } 1 \mid \{S_1; S_2\} \mid \textbf{skip} \mid \textbf{loop\_forever} \mid \textbf{choose } S_1 \textbf{ or } S_2$$

The print statement 'print 1' outputs a '1'. The sequencing statement, i.e, $\{S_1; S_2\}$, executes S_1 and S_2 in sequence. The skip statement **skip** does nothing, and the statement **loop_forever** performs an infinite loop. The simple non-deterministic statement **choose** S_1 **or** S_2, makes an arbitrary choice between statements S_1 and S_2. We will also allow procedure definitions such as

```
procedure p;
  S.
```

The procedure p defined above is called with the statement

$$p.$$

A real concurrent language should of course have a much richer set of constructs, but the constructs we have listed are sufficient for our examples. The following procedure definitions are intended to illustrate different aspects of the interplay between non-determinism and infinite computations.

```
procedure p;
  choose skip
  or     {print 1; p}

procedure q;
  choose skip
  or     {q; print 1}

procedure r;
  {print 1; r}
```

The body of procedure p consists of a non-deterministic choice which either does nothing, or outputs a 1 and then repeats. The behaviour of p can very naturally be illustrated as a tree where the nodes are non-deterministic choices and the edges output statements.

The body of procedure q is similar to the body of p. We have a non-deterministic choice, but in the second branch the recursive call comes before the print statement. This means that q cannot produce an output before the return of the recursive call, but at this stage all q has left to do is to execute a few more print statements. q can behave in two different ways. Either q will choose the second branch every time, in which case no output will be produced, or q will choose the first branch of the non-deterministic statement at the nth level of recursion, in which case it will produce a sequence of n '1'.

The procedure r is deterministic. It will produce an infinite sequence of ones.

Before we turn to the observable properties of these procedures, note that since our language allows a sequencing statement we must include termination in the set of observable properties to make the semantics compositional. We will indicate termination by a 't'.

We begin by looking at the observable properties of p, q and r, under the assumption that infinite observations are allowed.

$$\begin{array}{ll} p & 1^*t \cup 1^\omega \\ q & 1^*t \cup \epsilon \\ r & 1^\omega \end{array}$$

With infinite observations, we can easily determine the differences in behaviour between the three procedures.

Now, let us consider a semantics based on finite observations. We first consider as observables the set of results of terminated computations. The observable behaviour of the statements p, q and r with procedure definitions as above, are as follows.

$$\begin{array}{ll} p & 1^*t \\ q & 1^*t \\ r & \emptyset \end{array}$$

This is clearly not sufficient to give a meaningful semantics. When we only consider the behaviour of terminated computations, it is not possible to distinguish between the procedures p and q which clearly have differing behaviour. When we only consider observations made on terminating computations, infinite computations become completely invisible. This is not reasonable, since the output generated by an infinite computation is certainly visible.

Instead, let us consider a semantics of finite prefixes of arbitrary computations.

p	$1^*t \cup 1^*$
q	$1^*t \cup 1^*$
r	1^*

This is a slight improvement. Here we can see that the program r may generate output, but we can still not distinguish between p (which will either terminate or keep producing output) and q (which may perform an infinite computation without producing any output).

The traditional way to give a semantics based on finite observations which can distinguish between programs like p and q is to introduce the concept of *divergence*. A program is said to diverge if it can do an unbounded set of computation steps without producing output. Using a "d" to indicate that a computation is diverging, the observable behaviour of p, q and r is as follows.

p	$1^*t \cup 1^*$
q	$1^*t \cup 1^* \cup d$
r	1^*

(The only difference between this table and the previous is that we have added the information that q may diverge.) By the introduction of divergence as an observable entity, we can distinguish between q, which may enter an infinite loop, and p, which will always either terminate or generate output. However, the use of divergence does not give us any information about computations with infinite output. Consider, for example, the procedure p', defined as follows.

```
procedure p';
    choose p
    or     loop_forever
```

p' may either behave like p, or enter an infinite loop, so the observable behaviour of p' is $1^*t \cup 1^* \cup d$, that is, the same as that of q. However, p' and q differ in that p' may produce an infinite result, whereas q may not. When q starts to produce output, we know that q will eventually terminate.

The use of divergence as an observable gives us more information about the behaviour of a program, but we can not use divergence to distinguish between programs that only generate finite results, and programs that may generate infinite results.

The use of divergence as an observable property allows us to recognise the set of programs that always terminate, and give a reasonable semantics for such programs. However, allowing divergence as an observable property is questionable. To determine whether a program will diverge is equivalent to solving the unsolvable halting problem. Alternatively, one could see an observation of divergence as waiting forever to see whether the program will output anything, but the reason for introducing divergence was to avoid infinite observations, Another problem with the introduction of divergence as an observable is that we make the semantics more complex, since we add a new concept to the semantic model. Finally, divergence does not give information about infinite computations.

By allowing infinite observations of the behaviour of a computation, we get a number of advantages,

1. we can reason about the behaviour of infinite computations,
2. we avoid introducing in the set of observable behaviours behaviour that cannot be observed, and
3. the semantic model becomes simpler, since we have fewer concepts.

For the rest of this thesis, we will only consider semantics which allow infinite observations. We will not consider divergence.

1.6 Why external behaviour only?

The semantic models I present in the thesis are only intended to capture the external behaviour of programs. In other words, two programs which behave the same are seen as equivalent, even though they may differ in efficiency.

The decision to only consider external behaviour of programs is partly motivated by the interest in the meaning of programs. Suppose one modifies a program so it becomes slightly slower, but computes the same result. Has its meaning changed? I would argue that it has not.

The separation of efficiency and semantics can also be defended from a practical point of view, since it allows us to argue that one program has the same semantics as another, but is more efficient. Thus, one can, for example, motivate replacing a slow sorting routine with a faster one.

1.7 Goals

The goal of this thesis is to develop techniques to describe concurrent programming languages through denotational semantics according to the basic premises below.

Choice of formalism The formalism we base our investigations on should have the expressiveness of a normal programming language. That is, recursion should be allowed and it should be possible to write programs involving complex data structures. The language should also allow process creation and it should be possible to pass not only values but also channels between processes. At the same time, the formalism should have a simple formal definition. Given these requirements, the best choice is, in my opinion, concurrent constraint programming.

What types of computations to consider? An important underlying premise of this work is the notion that to give the semantics for a concurrent language it is necessary to concider infinite computations. We have already discussed these matters; let us just remind the reader that there are concurrent applications that are not intended to terminate.

The elements of semantic description Denotational models for concurrent programming in general have usually had domains which were based on communication events. This has made the semantic rules rather complex. Also, from a practical point of view; if we are to consider infinite computations even very simple programs may have to be described using uncountable sets of sequences of communication events. As a consequence, the semantic rules are not computable. It is certainly unattractive and unnatural to give the semantics of a programming language in terms of uncomputable functions.

The conclusion I draw is that we should strive toward semantic models which are based on abstract dependencies between input and output, and try to avoid models based on communication events.

1.8 Organisation of the thesis

Chapter 2 gives a historical overview of asynchronous concurrent programming. We describe various types of asynchronous concurrent languages, and give a brief survey of the central results concerning the semantic description of asynchronous concurrent languages.

Chapter 3 gives an informal introduction of ccp using a series of examples. The intention is both to give the reader an intuitive understanding of ccp, and to demonstrate the expressiveness of ccp.

In Chapter 4 we give a formal definition of ccp. We first give a general construction of constraint systems, and then point out a set of algebraic properties which we expect any constraint system to satisfy. Next, we formally define the set of ccp programs, and give a structured operational semantics which specifies one aspect of the operational behaviour. However, it is not possible to describe the set of fair computations using a structural operational semantics so we give a definition of fairness. We also review some results concerning closure operators, and the semantics of deterministic ccp programs. In the final section we give a simple semantics, which specifies the observable properties of a process, and the trace semantics, which gives more detailed information about the interactive behaviour of a process.

In Chapter 5 we give a fully abstract semantics for ccp. This semantic model is however not a fixpoint semantics. We give two proofs of full abstraction; one which depends on the use of an infinite conjunction of processes, and one which makes some assumptions about the constraint system. We also show some algebraic properties of ccp. It turns out that the semantic domain of the fully abstract semantics satisfies a number of algebraic properties. These properties correspond to the axioms of intuitionistic linear algebra, an algebra which was developed to model the properties of linear logic. Also, it is worth noting that the semantics of selection can be derived from other constructs.

Chapter 6 is an attempt to relate the results concerning the fully abstract semantics of ccp to corresponding results for data flow languages. We define a simple data flow language and give a fully abstract semantics for this language using the techniques described in Chapter 5.

Chapter 7 shows that for a large class of concurrent programming languages there is no denotational semantics which is a fixpoint semantics and fully abstract, and able to describe infinite behaviour. Similar results have been presented by other authors, but the result presented here is more general and can be applied to many different mathematical structures.

In Chapter 8 we introduce an operational semantics of ccp based on the use of *oracles*. An oracle describes the sequence of non-deterministic choices to be made by a process. One can either see an oracle as something that is given to a process right from the start of a computation, or as something that we extract from an existing computation. In the oracle semantics we can show two confluence properties. The first confluence property concerns finite sets of computations and finite sequences of computations steps and is similar to the confluence properties shown for other languages. The second confluence property concerns infinite sets of computations and infinite sequences of computation steps. As far as I know, no similar confluence property has been described for ccp or any other concurrent programming language.

Chapter 9 gives a fixpoint semantics for ccp, based on the oracle semantics. It turns out that even though the oracle semantics is compositional, the existential quantifier is not continuous under any ordering in which the other constructs of ccp are also continuous. Thus, we are forced to resort to a fixpoint semantics in which values of local variables of a process are made part of the semantics of the process.

To deal conveniently with the local variables of an agent we introduce a set of 'hidden' variables and a couple of renaming operations. We also show that the renaming operations satisfy a number of algebraic rules. The 'algebra of hiding' is quite general and it is possible that it may find other applications.

The definition of the fixpoint semantics is quite straight-forward. We show correctness, i.e., that for a given process the fixpoint semantics gives exactly the same set of traces as the fully abstract semantics. We also give a version of the fixpoint semantics in which the the semantic domain is a category.

Finally, Chapter 10 concludes with some remarks on the underlying assumptions of the thesis, the significance of the results, possible applications of the results, and future research.

1.9 Acknowledgements

I would first like to thank Bengt Jonsson, my thesis advisor, for discussions, helpful comments and encouragement.

I would also like to thank Håkan Millroth and Roland Bol who read an earlier version of the thesis, pointed out errors and requested clarifications.

During the early part of my thesis work Keith Clark gave me valuable help.

Thanks also to colleagues at the computing science department for many enlightening discussions.

Chapter 2

Models of asynchronous concurrent computing: an overview

This chapter is an attempt to put the thesis in its context. We give an overview of the history of concurrent computing—focusing on concurrent programming with *asynchronous* communication, and an overview of the work done on formal semantics for concurrent languages.

Asynchronous communication implies that the only synchronisation between processes is when a process waits for a message. In other words, if a process wants to send a number of messages to another process, it is allowed to do so, regardless of whether the receiving process reads the messages or not.

The first examples of asynchronous communication on a computer was the use of *buffering* between the central processing unit and various I/O devices (for example printers, card readers, tape stations) to improve the utilisation of the central processing unit. The use of buffered I/O implies that the program can send data to a device, even if the device is currently busy, and that a device can accept input, even if the central processing unit is busy.

The idea of using buffered communication between processes is mentioned by Dijkstra [26]. An early description of a programming system in which the communication between processes is done in a buffered, asynchronous manner was given by Morenoff and McLean in 1967 [55].

2.1 An early asynchronous programming system

In 1970, Brinch Hansen [8] presented an operating system which allowed dynamic creation and destruction of a hierarchy of processes. Communication between processes was done by assigning to each process a *message queue* for incoming messages. In the presentation below, *buffer* is a fixed-size memory area (eight 24-bit words) which is used to store a message.

A message queue can thus be represented as a linked list of buffers. The buffers are maintained in a pool by the operating system.

The communication primitives are

> send message (receiver, message, buffer)
> wait message (sender, message, buffer)
> send answer (result, answer, buffer)
> wait answer (result, answer, buffer)

Send message picks a buffer from the buffer pool and copies the message into the buffer. The buffer is then put into the receiver's message queue. The receiver is activated if it is waiting for a message. The sender revives the address of the buffer and continues its execution.

Wait message checks if there are any messages in the queue. If not, the process is deactivated until any message arrives. When a message arrives, the process is provided with the name of the sender and the address of the buffer.

When a process has received a message, and wants to reply to the original sender, the primitive *send answer* is used. The original message buffer is re-used, the new message is put into the buffer, the buffer added to the original sender's message queue, and the original sender activated if it is waiting for the answer.

A process that expects an answer to a message it has sent uses the command *wait answer* to delay until an answer arrives.

The first two primitives are the most interesting, the primitives 'send answer' and 'wait answer' can actually be implemented using 'send message' and 'wait message'. However, it appears that the situation when a process expects an answer to a message would be quite common in actual programs, so the inclusion of the two last primitives is probably well-motivated.

Note that in Brinch Hansen's model, non-determinism is implicit. For example, if two processes A and B each send a sequence of messages to a third process C, the messages received by C will be an interleaving of the messages sent by A and B. The relative order of the messages from A and B depends on the scheduling mechanism of the operating system and can not be determined by examining the program. However, the messages from A will be seen by C in the same order as they are sent from A, and the same holds for the messages sent by B. This is an example of a *non-deterministic merge*.

2.2 Kahn's fixpoint semantics

In 1974, Kahn [37] presented a fixpoint semantics for a simple parallel language. This work has had a profound influence on the design of parallel programming languages, and on the theory of parallel programming.

Kahn considered a parallel programming language based on a process model very similar to the one given by Brinch Hansen. The process definitions look like procedure definitions in an imperative programming language. Each procedure has a number of input channels and output channels. A program is a directed graph of processes and channels, each channel being the input channel of one process and the output channel of another. Some channels may be connected to the outside and receive data from some agent outside the program, or send data to the outside. Communication is done in a style fairly similar to Brinch Hansen's. There is a command *send* I *on* V which sends message I on channel V, and a function *wait*(U) which deactivates the process until data appears on channel U, and then returns that message. There is no provision for allowing a process to wait for incoming messages on more than one channel. Note that the possibility of having more than one input channel can easily be simulated in Brinch Hansen's model, since each message is identified by the name of its sender.

The central result of Kahn's work is that each process can be described as a function. Suppose that the messages $a_1, a_2, \ldots, a_n$ have arrived over a channel at a given point of time (that is, the sequence $a_1, a_2, \ldots, a_n$ is the history of the channel). Naturally $a_1 a_2 \ldots a_n$ forms a string. Initially, this string is empty, and during the execution of the program the string grows as messages are transmitted over the channel. A process can now be seen as a function from the histories of input channels to to the histories of output channels. If we order the strings representing possible histories of a channel in the prefix ordering (so that a string x is smaller than a string y if x is a prefix of y) we find that the functions are monotone. Further, if we want to model infinite computations (and Kahn and I agree that we should) we must extend the set of histories to also include infinite histories. The set of finite and infinite strings are ordered such that an infinite string is greater than each finite prefix of the string. Under this ordering each process can be modelled as a continuous function from input channels to output channels. A program is thus modelled as an equation system. The meaning of the program is then the function that takes an input history as input and returns the corresponding minimal fixpoint of the equation system.

It is important to note that the restrictions that only *one* process may output data on a particular channel, and that a process may only wait for incoming data on *one* channel mean that a computation is essentially *deterministic*; even though the scheduling of processes can give rise to different execution orders, the output produced by the program is still completely determined by the input.

Note that Kahn's semantics only considers the external behaviour. Concepts like termination, divergence and deadlock are not present. For example, suppose that we have a program A and create a program A' by adding a process which has no input or output channels. Clearly A and A' will have exactly the same communication patterns, regardless of the internal behaviour of the added process. We can also consider the case when we add a group of processes which are connected to each other but are not to connected to processes in A, or to the outside. Let us call this process A''. We can imagine that the new processes of A'' may engage in infinite communication sequences, or wait for a message that never arrives, or perform infinite computations without any communication actions, or terminate. In all these cases, the external behaviour of A'' is exactly the same of A, and the Kahn semantics of the two programs is also the same. (One can of course argue that A is more efficient than A'', but this difference in efficiency should not be refelected in the semantics.)

It should be noted that the idea of viewing a concurrent process as a function over streams had already been considered by Landin [44].

2.3 A note on terminology: Definitions of the Non-deterministic Merge

As we have already indicated, the non-deterministic merge operator plays an important role in the theory of concurrent programming. The basic idea is that the merge operator has two (or more) input streams, and one output stream. As tokens arrive at any of the input streams, they are copied to the output stream. The easiest way to describe the behaviour of the merge is to say that it outputs the tokens in the order at which they arrive, but we do not want to introduce timing details into the semantic describtion, since timing is implementation-dependent. There are many ways to formalise the intuitive idea of how a merge operator should behave. The merge operators decribed below are the most common ones in the literature. The terminology is quite standard.

The *fair merge* guarantees that each token that arrives on any of the input streams will appear on the output stream.

The *angelic merge* guarantees fairness in the case when all input streams are finite, i.e., when all input streams are finite each incoming token will appear on the output stream. The angelic merge also guarantees that all tokens that arrive at one input stream will be output, in the case when the other input streams are finite.

The *infinity-fair merge* guarantees fairness in the case when all input streams are infinite.

Among the merge operators, the fair merge is the strongest. The reason for considering the other merge operators is that they are in some contexts easier to describe and implement. The angelic merge can be expressed in a concurrent language with non-deterministic guarded choice, but we cannot express fair merge without making asumptions about the selection mechanism. An infinity-fair merge can be expressed as a deterministic procedeure that takes as input (beside the input streams to be merged) an oracle, which tells the merge from which input stream the next token should be read. Thus, the choices made by the infinity-fair merge are not dependent on input.

2.4 The data flow languages

In 1975, Dennis [25] presented a concurrent programming language (described as a data flow procedure language) with mechanisms for communication very similar to those described by Brinch Hansen and Kahn. The difference is that here the processes, or 'nodes' as Dennis calls them, are not complex procedures written in some imperative language, but instead very simple computational units. The language definition describes a number of types of nodes, and the idea is that the programmer should create a program by connecting a network of nodes. Even though this language from a pragmatic point view is very different from the ones which allow complex processes, it is still possible to apply Kahn's techniques.

After Kahn had shown how to give a fixpoint semantics for a deterministic data flow language several people tried to generalise the results to non-deterministic data flow languages. We will here review the results presented the years after Kahn's semantics had been presented.

Kosinski [41, 42] gave a fixpoint semantics for non-deterministic data flow. One of the central ideas in his approach was to associate with each token in a stream the sequence of non-deterministic choices which lead to the generation of the token. However, the semantics is rather complex and is difficult to understand. As Clinger [19, page 85] pointed out, one of the crucial theorems has a flaw in its proof.

Keller [39] discussed several approaches to the formal semantics of non-deterministic concurrent programming. He showed in an example that non-deterministic merge could not be modelled using simple input-output relations. A similar result was later given by Brock and Ackerman [9].

Broy [14] points out that an applicative language that has a parallel evaluation rule and is extended with McCarthy's ambiguity operator [51] is sufficiently powerful to implement the merge operator. He presents a rather complex fixpoint semantics based on an intricate powerdomain construction. The operational semantics of his language does not address fairness, so it is unclear how his semantics deals with infinite computations.

Park [63] considered a non-deterministic data flow language in which a program consists of deterministic nodes and merge nodes. He gave a fixpoint semantics in which non-determinism is modelled using oracles. Each merge node is provided with an extra argument, the oracle, which controls from which of the input streams the merge operator would read its next input. To cope with the case when one of the input streams is empty, Park introduced a special token τ, also called the hiaton, which would be emitted by each node at regular intervals, so that a node is never completely silent and there is always tokens arriving on each channel. A sequence containing τs is considered to be a representation of the same sequence with the τs removed. The model is not fully abstract, of course, but it is quite simple and easy to understand and it is easy to see precisely why it is not fully abstract. Also, the model deals with fairness and infinite computations.

Brock [9, 10] gave a fairly straight-forward semantics for non-deterministic data flow in which the semantic domain consisted of sets of *scenarios*. A scenario is a graph of input and output events, in which the graph structure records causality.

2.5 Concurrent programming with synchronous communication

The development of the synchronous concurrent languages CSP and CCS had two motivations.

First, it appeared difficult to give a satisfactory semantic description of the data flow languages, either by an operational definition or by a denotational semantics. In contrast, the operational semantics of CCS could be presented concisely, and for CSP there was at an early stage a fixpoint semantics [11].

Second, there is an obvious technical problem with the asynchronous communication approach. Since the number of messages sent but not received may be arbitrarily large, we need a mechanism for dynamic allocation of messages. If our programming language is a low-level language in which all memory is allocated statically, one might want to avoid introducing a dynamic memory allocation scheme just for the sake of handling the allocation of messages. So, if asynchronous communication implies unbounded buffers, what is the alternative?

In synchronous languages such as Hoare's CSP (communicating sequential processes) [31, 32] and Milner's CCS (calculus of communicating systems) [54] one can say that the communication channels have buffers of size zero. Since there is no way to store a message which has been sent but has not yet been read by the receiver, it follows that the sender cannot be permitted to send the message until the process on the other end of the communication channel is ready to receive. It follows that we need a synchronisation mechanism that prevents a process from sending a message until the receiver is ready to accept the message.

CSP as described by Hoare in the 1978 article [31] has two communication primitives, the output statement and guarded wait. The output primitive sends a message on a channel, and delays the process until the receiver accepts the message. The input primitive suspends the process until a message arrives. The input may occur in a guard, which implies that a process may wait for input on many channels, but also that a process may wait for a specific message, or a message that satisfies some given property.

Given asynchronous communication, as described in previous sections, it is straight-forward to implement the communication primitives of CSP according to the following scheme. The output statement of CSP is implemented as an asynchronous send followed by a statement that waits for the receipt of an acknowledgement. The receiver is then responsible for sending an acknowledgement back to the sender, after it has read the message. In the same way asynchronous communication can be implemented in a synchronous language using an explicit buffer.

The scheme to implement the synchronous communication mechanism using asynchronous send and acknowledgement could be used to implement CSP on a computer with distributed memory, but it also shows that the difference between synchronous and asynchronous communication is not a matter of profound theoretical importance, but rather one of convenience of programming and efficiency of implementation. One can argue that either of the two communication mechanisms is more primitive than the other, which one that ends up being more primitive would depend on the computational model one chooses for the comparison. In a comparison between CSP and Kahn's semantic model, Hoare characterises Kahn's model as a more abstract approach, while CSP is described as being more machine-oriented [31, page 676].

The communication primitives of the version of CSP presented by Hoare in his book [32] (sometimes referred to as theoretical CSP, or TCSP), and also of CCS [54], allow a more complex form of communication, in which both input and output statements may be guarded.

Brookes, Hoare and Roscoe [11] gave a fixpoint semantics for TCSP, in which only finite observations of communication actions were considered. As pointed out by Abramsky [1] there are programs which differ in their infinite behaviour, but still have identical finite behaviour, so clearly a semantic model based on finite observations cannot successfully treat infinite computations. If we consider *terminating* processes, it is easy to see that they allow the same set of finite observations exactly when they have same external behaviour. However, it is possible to find terminating and non-terminating processes which exhibit the same finite behaviour. Thus, we cannot tell whether a process is terminating by making finite observations.

The above holds if the only observations allowed are communication actions. Brookes and Roscoe [12] presented an improved fixpoint semantics in which termination and divergence (that a process enters an infinite loop without doing any communication) were made part of the externally visible behaviour.

2.6 The Committed choice and Concurrent logic programming languages

People in the logic programming community have during the last 20 years proposed a great number of logic programming languages with features that would allow parallel implementation, concurrent programming, or both. It is not possible to give an account of all the ideas people have been tossing around (and sometimes tried to implement). Instead, we will concentrate on a group of concurrent languages intended to have reasonably straight-forward and efficient implementations, and which have communication primitives which largely resemble the send and wait operations described above. The languages which belong to this group were initially referred to as the *committed choice* logic programming languages, to reflect that programs written in these languages will not backtrack, unlike Prolog programs. Later, the term 'concurrent logic programming languages' (clp) has also been used.

The committed choice languages result from attempts to develop a concurrent version of Prolog, suitable for parallel implementation. It seemed that conjunction, which is sequential in Prolog, but not in logic, could be seen as a parallel composition operator. A conjunction of goals could then be read as the creation of communicating processes.

It was clear that to execute goals in parallel, one would have to find a way to avoid problems with the back-tracking mechanism of Prolog, which may involve many goals. It is worth noting that the back-tracking mechanism of Prolog is seldom used to find more than one solution of a call; usually some back-tracking may occur in the selection of a clause and few programs take advantage of Prolog's search mechanism in more complex situations. Also, in most logic programs it is well-defined which arguments of a predicate are input and which ones are output, so the generality offered by Prolog is seldom used in actual programs.

The committed choice languages solved the problem with back-tracking by eliminating deep back-tracking altogether. The committed choice languages had the following characteristics. First, allow the goals of a conjunction to execute in parallel. Second, do away with general back-tracking. Instead, provide a programming construct that makes it explicit which tests should be satisfied before a clause of a predicate can be selected, i.e., *guards*.

In general, a clause in a committed choice language has the form

$$H : - G \mid B,$$

where H is the head of the clause, G is the guard consisting of zero or more tests, and B is the body of the clause, containing a number of calls. The idea is that for a given call, a clause can be selected if the unification of the head of the clause with the call succeeds, and the tests in the guard succeeds. Before a clause is selected, the head unification and the guard tests must not give any globally visible variable bindings. After a clause has been selected, the goal may not back-track and try other alternatives, i.e., the goal has committed to one alternative.

The main difference between the various committed choice languages lies in the means for making sure that the head unification and execution of guard tests is not visible to other goals (this is usually referred to as the synchronisation mechanisms).

Relational language (Clark and Gregory [17]) requires *mode declarations* for each predicate. A mode declaration contains information on which arguments of a predicate of are input and which are output. In a conjunction of goals, it is required that each variable occurs in an output position in at most one goal. In Relational Language it is possible to program a form of communication between processes which is very similar to the ones considered by Brinch Hansen, Kahn and Dennis, by using lists as the communication medium.

Shapiro's Concurrent Prolog [76] (CP) used a rather complex read-only variable annotation to control the directionality of bindings. As pointed out by Ueda [80] and Saraswat [70] the semantics for unification with read-only variables was not well-defined. Shapiro has presented a corrected version of Concurrent Prolog which has a well-defined semantics [74]. However, even the corrected version of Concurrent Prolog uses a complex unification mechanism which seems problematic, both for programmers and for implementers.

The article by Shapiro contained several inspiring program examples, and has been influential, despite the deficiencies of Concurrent Prolog. One important difference between Relational Language and Concurrent Prolog is that Concurrent Prolog allows a very appealing form of two-way communication. A message can contain an empty slot which the receiver of the message fills in, and the original sender reads. This is actually quite similar to Brinch Hansen's 'send answer' and 'wait answer' described earlier. The two-way communication allowed a number of interesting programming techniques. It is possible to program lazy evaluation, and one can also simulate objects with state.

In 1983, Clark and Gregory presented a new committed choice language called Parlog [18]. The communication mechanism differs from that of Relational Language, in that the directionality requirement of Relational Language is relaxed, and two-way communication as described by Shapiro is possible.

Guarded Horn Clauses was proposed by Ueda [81, 82] as a general-purpose programming language for the ICOT project. The language was intended for both sequential computers, and for multi-processor computers. The synchronisation mechanism was a simple rule, stating that the head unification and execution of goals in the guard must not affect the global state. The early description of Guarded Horn Clauses (GHC) allowed calls to user-defined predicates in the guard of a clause, but since there was no obvious efficient way to enforce that the execution of a goal in a guard did not export any bindings to the global state (however, see [57] for a possible way to implement this), later developments focused on a restricted form of GHC, called flat GHC (FGHC), in which the guard could only contain simple tests (such as $X > Y$). The ICOT project decided to use a version of FGHC as implementation language. This language became known as KL1 (Kernel Language 1).

Strand [27, 28] is a further restriction of FGHC, in which general unification has been replaced by a simple assignment operation, which requires one of the arguments to be an unbound variable. There is an implementation of Strand that allows Strand programs to be interfaced with routines written in conventional languages, such as C and Fortran. This makes it possible to use Strand as a coordination language allowing the development of efficient parallel programs for multi-processor machines.

The class of concurrent logic programming languages consists of languages where the execution model is based on resolution together with some mechanism for synchronisation between goals. These basic notions are sufficient to make the clp languages flexible and expressive concurrent programming languages. It is perhaps unfortunate that so much effort went into the development of complicated synchronisation mechanisms (the review on these pages has merely scratched the surface), when very simple synchronisation rules are sufficient to make the clp languages as powerful as most concurrent programming languages. To get an idea of the wealth of synchronisation mechanisms developed for concurrent logic languages, the reader is directed to Shapiro's survey [75].

2.7 Concurrent Constraint Programming

The definition of the various committed choice languages involved both a description of the operational behaviour of the different constructs, and a specification of the data structures (terms) that the programs would operate on. As an alternative to the committed choice languages, concurrent constraint programming was proposed by Maher [48] and Saraswat [71]. They showed that it was possible to gain clarity by separating the issues regarding the operational properties of the language from the data structures. The result is a formalism, which is suitable for a formal treatment and allows a simple operational description.

Concurrent constraint programming can be seen as a generalisation of committed choice languages such as GHC and Strand, as well as of the data flow languages and of the applicative concurrent languages.

2.8 Semantics for concurrent logic and concurrent constraint languages

A number of semantic models for various concurrent logic languages have been proposed. Those mentioned below all consider only finite computations, unless stated otherwise.

To give a semantics for a concurrent logic or concurrent constraint language is not very different from giving the semantics for other concurrent languages. The feature unique to the clp languages is that the synchronisation rules involve terms and substitutions. In concurrent constraint programming, these operations have been 'abstracted away' and what remains are operations that can be seen as basic lattice-theoretic operations. Many results and techniques for other concurrent languages are also applicable to the clp languages and ccp.

In ccp and also in some clp languages such as Parlog and GHC, the guards are monotone, which means that if the tests in a guard are true at one point in time, they will remain true in the future. This property simplifies the implementation on some types of multi-processors, and can also influence the semantic description.

De Bakker and Kok gave an operational and a fixpoint semantics for a guarded subset of Concurrent Prolog and proved the equivalence between the two semantics [21]. The domain was a metric space in which the elements were trees of program states, and the reason the semantics only considered guarded programs was to make sure that the corresponding semantic functions would be contracting, thus guaranteeing unique fixpoints.

Saraswat, Rinard and Panangaden [71] gave several semantic models for concurrent constraint programming. These included a fully abstract fixpoint semantics for deterministic concurrent constraint programming. This model is a generalisation of Kahn's semantics. They also gave a model for nondeterministic concurrent constraint programming, in which only finite computations were considered. This model was based on a semantics presented by Josephs [36] and is related to Brookes and Roscoe's model mentioned earlier.

De Boer and Palamidessi [23] also presented a fully abstract semantics for concurrent constraint programming, but also in this case only for programs that only exhibit finite behaviours. In their paper, De Boer and Palamidessi formulate the fully abstract semantics by means of structural operational semantics.

De Boer, Kok, Palamidessi and Rutten [22] give a general framework for the semantics of concurrent languages with asynchronous communication. They note that the traditional failures model [11] is unnecessarily detailed when applied to programming languages which only allow asynchronous communication.

2.9 Developments in the semantics of concurrency

In the early literature on concurrency, a program was said to exhibit *unbounded nondeterminism* if it was guaranteed to terminate but could produce infinitely many different results.

Park [64] showed that for a non-deterministic imperative programming language with a fairness property it is possible to write a program that exhibits a form of unbounded non-determinism (see also an example in Section 3.15). Broy [14] gave a similar program for an applicative concurrent language.

Apt and Plotkin [4] considered an imperative programming language in which the control structures were if-then-else and while-loops. The language had a single non-deterministic construct, the *non-deterministic assignment*. The non-deterministic assignment is written $x = ?$, for some variable x, and may bind x to any integer. This type of non-determinism, in which it is possible to write a program that will always produce a result, but where there is an infinite set of possible results, is called *unbounded non-determinism*. Apt and Plotkin showed that for their programming language there could be no *continuous* fully abstract fixpoint semantics. However, they were able to give a fully abstract least fixpoint semantics by giving up the requirement that the semantic functions should be continuous. To see how this is possible, note that by the Knaster-Tarski theorem any monotone function over a complete lattice has a least fixed point, which implies that to define a fixpoint semantics it is sufficient to have semantic functions that are monotone.

Abramsky [2] studied a simple non-deterministic applicative language and showed that there could be no continuous fully abstract fixpoint semantics. As an alternative, he suggested the use of a categorical powerdomain. In reference [1], Abramsky showed that to give a meaningful semantics for a programming language with infinite computations, it is necessary to allow infinite experiments.

Kok [40] and Jonsson [34] have presented fully abstract semantics for non-deterministic data flow. Kok's semantics is based on functions from sequences of input sequences to sets of sequences of output sequences. Jonsson's semantics is based on *traces*, that is, sequences of communication events. The semantic model can express fairness properties and infinite computations but is not a fixpoint semantics and can thus not give the semantics for recursive programs. In a subsequent paper [35] Jonsson showed that the trace semantics was fully abstract with respect to the history relation.

Stoughton [78] presented a general theory of fully abstract denotational semantics. He gave a simplified version of Apt and Plotkin's negative result concerning the existence of continuous fully abstract fixpoint semantics for a language with unbounded non-determinism, and a negative result for languages with infinite output streams, based on Abramsky's proof.

Stark [77] gave a fixpoint semantics for a non-deterministic data flow language with infinite computations. The behaviour of a non-deterministic data flow network is represented by a function, where each maximal fixpoint corresponds to a possible behaviour.

Brookes [13] gave a fully abstract semantics for an imperative concurrent language where variables were shared between processes. In the model infinite computations and fairness were considered, but recursion and local variables of processes were not treated.

Panangaden and Shanbhogue [62] examined the relative power of three variants of the merge operator. They showed that fair merge can not be expressed using angelic merge, and that angelic merge cannot be expressed using infinity-fair merge.

2.10 Erlang

As an example of a recent concurrent programming language we give a brief description of Erlang.

Erlang [5] is an asynchronous concurrent programming language intended for real-time applications, in particular, applications in telecommunications. Important goals in the design of Erlang have been robustness and the ability to update code in a running system without stopping it.

Like in Brinch-Hansen's communication model, a program consists of relatively complex sequential processes, each with a single input channel. However, in Erlang the individual processes are programmed in a simple functional language, while the processes in Brinch Hansen's model could be programmed in any language that was available on the machine.

To allow real-time programming, Erlang has a timeout mechanism which allows the programmer to specify how long a process should wait for a message to arrive before doing something else. Another feature, apparently not present in Brinch Hansen's model, is the ability to scan the buffer of incoming messages for a message that matches a given pattern.

A primitive of Brinch Hansen's system, not present in Erlang, is the send-answer, wait-answer mechanism, which allows a process to directly reply to a message. It is possible to get the same effect in Erlang by a common programming idiom. First, a process that wants an answer to its message includes its process identifier (which gives the address of its input channel) in the message. The receiving (and answering) process can then extract the process identifier from the message, and send the reply to the original sender. To make it possible for the original sender to recognise the answer, the answering process includes its own process identifier in the message. Finally, the original sender can find the reply in the input channel by scanning it for a message matching a pattern, which includes the process identifier of the original receiver.

Even though the technique described above makes it possible for Erlang processes to answer messages, it appears that Erlang would benefit from a send-answer, wait-answer mechanism such as the one in Brinch Hansen's model, since this would mean a slight simplification to many programs, and it could also be more efficient than the present way of having a process answer a message.

2.11 Mechanisms for Communication

We conclude by giving an overview of the communication models mentioned in this chapter.

2.11.1 Asynchronous communication

The traditional asynchronous communication model is the one used by the data flow languages. The communication medium is buffered streams. A process may insert a message into a stream at any time, and the receiver will, when it attempts to read a message from a stream, either succeed, (there is a message) or suspend until a message arrives (if the buffer did not contain any messages).

One can see the send operation as adding information to the history of the stream, i.e., "the nth element of the stream is x", and similarly the read operation as asking question about the history of the stream "what is the nth element of the stream?". The generalisation to concurrent constraint programming is immediate, so we include ccp in this group, and also concurrent logic languages such as GHC.

2.11.2 Asynchronous communication with replies

A slight improvement of the asynchronous communication model is to make it possible for a receiver of a message to reply to it (this assumes that the sender expects an answer, of course).

There are many applications where it is natural to let a process respond to a message directly. For example, consider

1. a process maintains a state of some sort (anything between a simple counter and a huge database) and other processes request information about the state, and

2. a process is requested to do something and say when it is finished.

As described by Shapiro [76], asynchronous communication with replies can be directly expressed in a concurrent logic language. The techniques for doing this can be carried over directly to concurrent constraint programming.

2.11.3 Synchronous communication

The simplest form of synchronous communication as described in Hoare's CSP article [31] is as follows.

When a process sends a message the process is suspended until the receiver has read the message. When a process tries to read a message it is suspended until a message arrives.

The main advantage of synchronous communication over asynchronous communication is that no memory space needs to be allocated for buffers. Hoare showed how asynchronous communication could be emulated by synchronous communication using an explicit buffer. On the other hand, synchronous communication can easily be emulated by asynchronous communication if there is a mechanism for replies. In that case, we simply adopt the convention that each synchronised message should be implemented as a asynchronous send plus a reply (acknowledgement) by the receiver.

One example of a concurrent language that uses synchronous communication is Occam [50].

2.11.4 Synchronous communication by mutual knowledge

In the variant of CSP examined in different theoretical settings (theoretical CSP) it is allowed to put an output statement in guards. A guarded output statement succeeds if the receiver accepts the message, if the message is not accepted some other branch may be selected. This means that for a communication to take place between two processes, it is necessary that

1. the sending process *knows* that the receiving process is willing to take the message (if not, the sender may choose to do something else), and

A	A(r)	S	S(bmk)	S(bmk+r)
data flow	BH			
RL		CSP	CCS	
	Parlog	Occam	TCSP	CP
	GHC			
Erlang	CCP			

Table 2.1: Concurrent languages by communication mechanism. A, asynchronous communication; S, synchronous communication; (r), allowing replies to messages; (bmk), by mutual knowledge; BH, Brinch Hansen's system; RL, Relational Language.

2. the receiving process *knows* that the sending process is willing to send the message (if not, the receiver may choose to do something else).

In the above, doing something else may involve, for both the sender and the receiver, sending or receiving a message to another process.

There are indeed some concurrent programs that are easier to write using synchronous communication by mutual knowledge. However, synchronous communication by mutual knowledge is difficult to implement efficiently, in particular on a multiprocessor computer with distributed memory, and it is questionable whether the added expressiveness is really worth the additional complications in the implementation.

2.11.5 Shared memory

This is when processes share memory space or a portion of it. The shared memory communication model is the oldest type of communication between processes, and probably also the most common.

Even though in concurrent constraint programming processes may operate on a shared data structure, we still do not consider ccp to belong to this group since in ccp there is no destructive update of data structures.

2.11.6 Classifying message-passing languages

We have collected the concurrent languages mentioned in this chapter in Table 2.1 organised by communication method and (roughly) by year of introduction. Note that one programming language, Concurrent Prolog (CP) allows both replies and synchronisation by mutual knowledge.

Chapter 3

Concurrent Constraint Programming: Examples

We give some examples of concurrent constraint programs, to help the reader get an intuitive understanding of ccp. Some of the examples are accompanied by the same program in concurrent logic programming (clp) notation, since the notation of clp is sometimes more readable. For bigger and more interesting examples, the reader is referred to the textbooks written on concurrent logic programming, see for example Foster and Taylor's introduction to Strand [28].

In the examples, we will say that a variable is *bound* to a value if the constraints in the store imply that the variable should have that particular value. We will use capital letters for variables.

3.1 Constraint systems: the term model

Theoretically, the constraint system could be any logical structure. However, we want the constraint system to be efficiently implementable, and at the same time sufficiently powerful to make ccp an interesting programming language. In the examples in this chapter we assume a constraint system sometimes referred to as the *term model*. The term model, which corresponds to the data structures of Prolog, makes ccp about as expressive as concurrent logic programming languages such as Parlog, GHC or Strand.

Suppose we have a set of variables $\{X, Y, \ldots\}$, a set of function symbols $\{f, \ldots\}$, and a set of constant symbols $\{a, \ldots\}$. The set of terms is then inductively defined, as follows.

1. X is a term, if X is a variable.
2. a is a term, if a is a constant symbol.
3. $f(E_1, \ldots, E_n)$ is a term, if $E_1, \ldots, E_n$ are terms.

A constraint is of the form

$$E_1 = E_2,$$

where E_1 and E_2 are terms.

The intention is that the equality constraint is to be implemented using something called *unification*. Let me give some examples.

If the store contains

$$X = 3$$

we have, naturally, that the constraint $X = 3$ is entailed by the store, i.e., that X is bound to 3.

If the store contains

$$X = f(Y) \qquad \text{and} \qquad Y = 7,$$

we have that $X = f(7)$ is entailed by the store. (The constraints $X = f(Y)$ and $Y = 7$ are also entailed by the store, as is $Z = Z$, for any variable Z).

If the store contains

$$X = f(3) \qquad \text{and} \qquad X = f(Y),$$

it follows that the constraint $Y = 3$ is entailed. Thus, the terms are at the same time syntactic and semantic. If we know that X is bound to a term of the form $f(\ldots)$, we can add a constraint $X = f(Y)$ to the store, where Y is a fresh variable, to extract the contents of the term.

3.2 Agents

The basic programming construct in ccp is the agent. The simplest agent is the *tell constraint*, which is written as the constraint itself. For example, the agent (and tell constraint)

$$X = 5$$

adds the constraint $X = 5$ to the store. We can also say that the agent binds X to 5.

3.3 Ask constraints and selections

A *selection* is a sequence of pairs of the form

$$c \Rightarrow A,$$

where c is a constraint, i.e, an ask constraint and A is an agent. For example, the selection

$$(X = 3 \Rightarrow Y = 5)$$

checks if the constraint $X = 3$ is entailed by the store, and if it is, adds the constraint $Y = 5$ to the store. If the constraint $X = 3$ is *not* entailed by the store, the selection remains passive until the constraint becomes true.

3.4 Message Passing

As an example of communication between agents, consider the following agents.

$$X = 10 \qquad \text{and} \qquad (X = 10 \Rightarrow Y = 11).$$

Running the first agent will cause the constraint $X = 10$ to be added to the store. If we run the second agent, it will check that the constraint $X = 10$ is entailed by the store. If and when this is the case, the agent will add the constraint $Y = 11$ to the store. If we want to run the two agents concurrently, we can do this by putting them together in a conjunction.

$$X = 10 \wedge (X = 10 \Rightarrow Y = 11)$$

3.5 Hiding

If we want to enable two agents to communicate through a private channel, we can do this using the existential quantification. For example, if we write

$$\exists_X (X = 10 \wedge (X = 10 \Rightarrow Y = 11)),$$

we hide any references to the variable X made by agents *inside* the quantifier from agents on the outside (and also the other way around). When we run the agent above, the only detectable result is that the constraint $Y = 11$ is eventually added to the store.

3.6 A Procedure Definition

As a simple example, we give a boolean inverter.

$$\text{not}(X, Y) :: (X = 0 \Rightarrow Y = 1 \;[]\; X = 1 \Rightarrow Y = 0)$$

For example, a call $\text{not}(1, Y)$ will cause the variable Y to be bound to 0. A call $\text{not}(X, Y)$ will wait until X is bound. If X is never bound, the call will have no effect.

3.7 A Recursive Definition

Let us consider the traditional example of a recursive program, the factorial function.

$$\begin{array}{l}
\text{fak}(X, Y) :: \\
\quad (X = 0 \Rightarrow Y = 1 \\
\quad [] \; X > 0 \Rightarrow \exists_Z \exists_{X_1} (X_1 = X - 1 \\
\qquad\qquad\qquad\qquad \wedge\, \text{fak}(X_1, Z) \\
\qquad\qquad\qquad\qquad \wedge\, Y = Z * X))
\end{array}$$

A call $\text{fak}(X, Y)$ will bind Y to $X!$, when X is bound to a positive integer.

3.8 Lists

We will use the same syntax for lists as Prolog. [] is the empty list, $[X \mid Y]$ is a list where the first element is X and the rest of the list is Y. A list of n elements can be written $[X_1, X_2, \ldots, X_n]$, where X_1 is the first element and so on. So, if $Y = [2,3]$ and $X = 1$ it follows that $[X \mid Y] = [1 \mid [2,3]] = [1,2,3]$.

As an example of a program that generates lists, we give the following definition. The procedure creates a list of length N that contains ones.

```
how_many_ones($N, X$) ::
            ($N = 0 \Rightarrow X = [\,]$
            [] $N > 0 \Rightarrow \exists_{X_1}\exists_{N_1}(X = [1 \mid X_1]$
                                  $\wedge\ N_1 = N - 1$
                                  $\wedge$ how_many_ones($N_1, X_1$)))
```

A call how_many_ones$(3, X)$ will generate a sequence of successive stores. The successive stores will contain stronger and stronger constraints for the value of X, as indicated by the sequence of constraints below.

$$\begin{array}{r} \exists_Y (X = [1 \mid Y]) \\ \exists_Y (X = [1,1 \mid Y]) \\ \exists_Y (X = [1,1,1 \mid Y]) \\ X = [1,1,1] \end{array}$$

Note that the list is generated from the head, and that thus the initial part of the list is accessible before the list is completely generated.

3.9 A non-deterministic program

In all examples above, the selections have been written so that no more than one condition in a selection can be selected, for a given store. A selection which satisfies this property is *deterministic*.

As an example of a program that contains a selection that is *not* deterministic, consider the following.

$$\text{erratic}(X) :: (\mathbf{true} \Rightarrow X = 0 \;[]\; \mathbf{true} \Rightarrow X = 1)$$

(Here we use **true** as a shorthand for some arbitrary constraint that always holds, like $X = X$.) The agent erratic(X) will either bind the variable X to 0 or to 1. It is important to keep in mind that the programmer cannot assume the procedure to satisfy any probabilistic properties or fairness conditions; for example, an implementation where the agent always binds X to 0 is correct.

3.10 McCarthy's ambiguity operator

We can give a more interesting example of a non-deterministic procedure definition.

$$\text{amb}(X, Y, Z) :: (\text{number}(X) \Rightarrow Z = X \;[]\; \text{number}(Y) \Rightarrow Z = Y)$$

The amb procedure, which is inspired by McCarthy's ambiguity operator [51] waits until either the first or the second argument is instantiated to a number, and then sets the third argument equal to the one of the two first that was defined. If both the first and the second arguments are numbers, the choice is arbitrary. (We assume that there is a predicate 'number' that holds for numbers and nothing else.) Consider the agent

$$X = 5 \wedge \text{amb}(X, Y, Z).$$

When the agent is run, the final store might be

$$X = 5 \wedge Z = X.$$

Of course, if the agent above is put into a context where Y is bound to a number, it is possible that the final store has Z bound to Y.

3.11 Merge

A slight generalisation of the 'amb' procedure.

```
merge($X, Y, Z$) ::
    ($\exists_A \exists_{X_1}(X = [A \mid X_1])$
         $\Rightarrow \exists_A \exists_{X_1} \exists_{Z_1}(X = [A \mid Z_1]$
                        $\wedge\, Z = [A \mid Z_1]$
                        $\wedge\, \text{merge}(X_1, Y, Z_1))$
    $[]\; \exists_A \exists_{Y_1}(Y = [A \mid Y_1])$
         $\Rightarrow \exists_A \exists_{Y_1} \exists_{Z_1}(Y = [A \mid Z_1]$
                        $\wedge\, Z = [A \mid Z_1]$
                        $\wedge\, \text{merge}(X, Y_1, Z_1))$
    $[]\; X = [\,] \Rightarrow Z = Y$
    $[]\; Y = [\,] \Rightarrow Z = X$)
```

A call merge(X, Y, Z) will, if X and Y are bound to finite lists, bind Z to an interleaving of the two lists.

3.12 The 'ones' program

To give an example of a program that produces an infinite result, we consider the simplest possible.

$$\mathrm{ones}(X) :: \exists_Y (X = [1 \mid Y] \wedge \mathrm{ones}(Y))$$

Running an agent ones(X) should generate longer and longer approximations of an infinite list of ones, i.e., stores where constraints of the form

$$\exists_Y X = [\underbrace{1, 1, \ldots, 1}_{n \text{ times}} \mid Y]$$

are entailed, for increasingly larger n. The 'final' result is then the limit of these stores, i.e., the store in which the constraint

$$X = [1, 1, 1, \ldots]$$

is entailed.

It is worth considering what happens when we have several agents that produce infinite results. Consider, for example, an agent

$$\mathrm{ones}(X) \wedge \mathrm{ones}(Y).$$

Clearly, we want this agent to bind both X and Y to infinite lists of ones. In other words, an implementation that ignores one agent and executes the other is incorrect. In Section 4.5 we will give a formal definition of fairness.

3.13 Two-way communication: the 'lazy-ones' program

All programs we have shown so far could have been written in most asynchronous concurrent programming languages. However, concurrent constraint languages and concurrent logic languages allow a kind of two-way communication, which increases the expressiveness.

We first consider a program that generates a list of ones. However, this program will only produce a one when requested.

We begin by giving the program in the form of a concurrent logic program, since the selection mechanism of clp is more readable in this case, and we expect that some readers may be more familiar with clp.

$$\begin{array}{l} \mathrm{lazy\_ones}([A \mid X_1]) :- \\ \quad A = 1, \\ \quad \mathrm{lazy\_ones}(X_1). \\ \mathrm{lazy\_ones}([\,]). \end{array}$$

In clp, a clause may be selected if the structures occuring in the head of the clause can be matched against the input. Given a call lazy_ones(X), the first clause can be selected if X is bound to a list of at least one element, and the second clause if $X = []$.

In the corresponding ccp program, the matching of input arguments is performed by ask constraints which contain existentially quantified variables. In general, to inquire whether X is of the form $f(Y)$, for some Y, we use the ask constraint $\exists_Y(X = f(Y))$.

```
lazy_ones(X) ::
    (∃_A ∃_{X_1} (X = [A | X_1])
        ⇒ ∃_{X_1} (X = [1 | X_1] ∧ lazy_ones(X_1))
     [] X = [ ]
        ⇒ true)
```

The first condition in the selection tests if X is bound to a list of at least one element. If this is the case, and the corresponding branch is taken, a constraint saying that the first element of X is a one will be added to the store. Then 'lazy_ones' is called recursively with the rest of the list as an argument. The second condition is for the case when X is bound to the empty list. The corresponding branch is the empty constraint **true**, i.e., when $X = []$ the agent lazy_ones(X) will terminate.

Now, consider an execution of the call lazy_ones(X). Suppose that X is unbound, i.e., that there is no information about X in the store. Neither of the two conditions hold, so the call cannot execute. Suppose now that X is bound to the list $[A_1, A_2 \mid X_1]$, for variables A_1 and A_2. Now it holds that X is a list with at least one element and thus the first alternative may be selected. Then A_1 is bound to 1, 'lazy_ones' is called recursively with $[A_2 \mid X_1]$ as argument, A_2 is bound to 1 and then the agent sits down and waits for X_1 to be bound.

The 'lazy_ones' program is of course not a very interesting program in itself, but the technique of using a partially defined structure to allow communication between processes has many possible applications. Shapiro [76] shows how it is possible to write concurrent logic programs in an 'object-oriented' style, where an object is represented as an agent which has a local state and reads and responds to a stream of messages.

3.14 Agents as Objects with State

Even though concurrent constraint programming does not allow destructive assignment, it is still possible to have objects with state. Below we give an example of a program which implements a stack as an agent which reads a stream of messages and responds to them (Shapiro [76]). We first give the stack program in clp notation.

$$
\begin{array}{l}
\text{stack}([\text{push}(X) \mid C_1], S) :- \\
\quad S_1 = [X \mid S], \\
\quad \text{stack}(C_1, S_1)). \\
\text{stack}([\text{pop}(X) \mid C_1], S) :- \\
\quad S = [X \mid S_1] \\
\quad \text{stack}(C_1, S_1)) \\
\text{stack}([\,], \_)
\end{array}
$$

The same program in ccp form.

$$
\begin{array}{l}
\text{stack}(C, S) :: \\
\quad (\exists_{C_1} \exists_X (C = [\text{push}(X) \mid C_1]) \\
\qquad \Rightarrow \exists_{C_1} \exists_X \exists_{S_1} (C = [\text{push}(X) \mid C_1] \\
\qquad\quad \wedge\, S_1 = [X \mid S] \\
\qquad\quad \wedge\, \text{stack}(C_1, S_1)) \\
\quad [] \; \exists_{C_1} \exists_X (C = [\text{pop}(X) \mid C_1]) \\
\qquad \Rightarrow \exists_{C_1} \exists_X \exists_{S_1} (C = [\text{pop}(X) \mid C_1] \\
\qquad\quad \wedge\, S = [X \mid S_1] \\
\qquad\quad \wedge\, \text{stack}(C_1, S_1)) \\
\quad [] \; C = [] \Rightarrow \textbf{true})
\end{array}
$$

A call stack$(C, [])$ will expect C to be bound to a list of commands. Suppose that C is bound to the list $[\text{push}(3), \text{push}(5) \mid C_1]$. The call to 'stack' will recurse twice and then suspend on the variable C_1. The internal state of the call, i.e., the second argument, i.e., the stack, is now the list $[5, 3]$. If C_1 is now bound to the list $[\text{pop}(X) \mid C_2]$, the top element of the stack (5) will be popped off the stack and set to be equal to X.

So, a call stack$(C, [])$ creates a stack object where the command stream C serves as a reference to the object. To allow more than one reference to the stack object we must use a merge agent and write, for example,

$$\text{stack}(C, []), \text{merge}(C_1, C_2, C)$$

to allow two agents to communicate with the stack.

3.15 Unbounded Nondeterminism

The last example is inspired by Park [64] and Broy [14], who showed that in a language with non-determinism and some fairness notion it was possible to write a program that exhibits unbounded non-determinism. We assume that the constraint system contains the natural numbers in the domain of values, and that constraints of the forms $X = 0$ and $X = Y + 1$ are allowed.

$$\begin{aligned}
p(X) :: {} & \exists_A A = 0 \\
& \wedge p_1(A, X) \\
p_1(A, X) :: {} & \exists_{A_1} \exists_{X_1} A_1 = A + 1 \\
& \wedge p_1(A_1, X_1) \\
& \wedge \mathrm{amb}(A, X_1, X)
\end{aligned}$$

In the successive recursive calls to p_1, the first argument will always be bound to an integer. Thus, the call to amb is guaranteed to terminate, and each recursive call to p_1 will bind its second argument to an integer. Clearly one possible result of a call of the form $p_1(n, X)$, where n is bound to an integer is to bind X to n. However, noting that a call $p_1(n, X)$ will result in the execution of $p_1(n+1, X_1) \wedge \mathrm{amb}(n, X_1, X)$ we see that it is possible for the recursive call to bind X_1 to $n+1$. Thus, the call to amb may bind X to $n+1$. It follows by an inductive argument that a call $p_1(n, X)$ may bind X to any integer greater or equal to n. Thus a call $p(X)$ will always bind X to an integer, and may bind X to any integer greater or equal to zero.

3.16 Remarks

As the reader probably has noted, the syntax of concurrent constraint programming is a bit unwieldy. Temporary variables for intermediate results have to be introduced, and what would have been a nested expression in a functional language is here a conjunction of agents, together with an existential quantification for the temporaries. That the syntax of ccp (and also of clp) makes programs less compact than they could have been is not a serious problem for program development, but it is of course a bit inconvenient when one wants to give program examples in a text.

Experience in concurrent logic programming shows that a more important problem with the syntactic form of programs (these concerns should also apply to ccp programs) is that simple programming mistakes, such as giving arguments to a call in the wrong order, or misspelling a variable, result in errors that are very difficult to locate. This is of course related to the difficulties of debugging concurrent programs, but a syntax that makes it easier to detect simple programming errors would be helpful.

However, these issues with syntax are not really relevant for us. I have chosen concurrent constraint programming as a vehicle for the investigations in the semantics of concurrency because of its generality and simple formal definition. As the examples in this chapter have shown, ccp is a powerful concurrent programming formalism that can emulate data flow, functional programing, and an object-oriented programming style. In the next chapter, we will look at the formal definition of concurrent constraint programming.

Chapter 4

Concurrent Constraint Programming: A Formal Definition

In this chapter, we give a formal definition of concurrent constraint programming (ccp). First, we formalise the concept of constraints. Second, we give the syntax of ccp programs, and third, the operational semantics of ccp programs. The operational semantics is not in it self sufficient to describe the behaviour of infinite computations, so as a fourth point we define a fairness concept.

Before we turn to the formal definition of ccp we mention some mathematical concepts which we will use in the following.

4.1 Mathematical Preliminaries

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 $f(x) \leq f(y)$. For a preorder $(L, \leq)$ and $S \subseteq L$, let $S^u = \{x \mid y \in S, x \leq y\}$.

A *partial order* is a pre-order which is also antisymmetric. 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 set $R \subseteq L$ is *directed* if every finite subset of R has an upper bound in R. 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 $\mathcal{K}(L)$ be the set of finite elements of L. A complete lattice L is *algebraic* if $x = \bigvee\{y \in \mathcal{K}(L) \mid y \leq x\}$ 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 $\mathcal{K}(L)$ we can easily extend f to a continuous function f' over L with $f'(x) = f(x)$, for $x \in \mathcal{K}(L)$, and $f'(x) = \bigvee\{f(y) \mid y \in \mathcal{K}(L), y \leq x\}$, for $x \in L \setminus \mathcal{K}(L)$.

For lattices $(L_1, \sqsubseteq_1)$ and $(L_2, \sqsubseteq_2)$ a pair of functions $f : L_1 \to L_2$ and $g : L_2 \to L_1$ is a *Galois connection between* L_1 *and* L_2 iff

1. f and g are monotone, and
2. for all $x \in L_1$, $x \sqsubseteq_1 g\,f\,x$, and for all $y \in L_2$, $f\,g\,y \sqsubseteq_2 y$.

The function f is called the *lower adjoint* and g the *upper adjoint* of the Galois connection.

Given a monotone function $f : L_1 \to L_2$ which preserves least upper bounds, we can construct a Galois connection (f, g) by $g\,y = \bigsqcup\{x \mid fx \sqsubseteq y\}$.

4.2 Constraints

4.2.1 Constraint Systems

A constraint system consists of logical formulas, and rules for when a formula is entailed by a set of formulas, i.e., a store. We assume that a set of formulas in a store is represented as a conjunction of the formulas. Thus, the logical operations we have to reason about are conjunction, existential quantification (to deal with hiding) and implication (entailment).

To define a semantics for concurrent constraint programming, we need a method to find a mathematical structure which contains the desired formulas, and also satisfies the properties which are needed to apply the standard techniques of denotational semantics. It is necessary that the constraint system is *complete*, that is, for a chain of stronger and stronger constraints there should be a minimal constraint that is stronger than all constraints in the chain. We also like the basic operations of the constraint system, the existential quantification, conjunction and implication, to be continuous. Palmgren [60] gives a general method to construct a complete structure from an arbitrary structure so that the formulas valid in the constructed structure are exactly those that are valid in the original structure. However, in this thesis we use a simpler construction which directly gives a complete constraint system.

To get the appropriate constraint system we start with a set of formulas, closed under conjunction and existential quantification, 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 [30] that do not involve negation.

In contrast, Saraswat *et al.* [71] choose an axiomatic approach, based on axioms from cylindric algebra and techniques from Scott's information systems [72] to specify the properties of a constraint system.

The use of ideal completion to construct a constraint system that is closed under infinite limits has previously been employed by Carlson [16] and Kwiatkowska [43].

Definition 4.2.1 A *pre-constraint system* is a tuple $\langle F, \mathit{Var}, \models, C\rangle$, where F is a countable set of *formulas*, Var is an infinite set of *variables*, C is an arbitrary set (*the domain of values*), and $\models\, \subseteq \mathit{Val} \times F$ is a *truth assignment*, where Val is the set of *assignments*, i.e., functions from Var to C. The only assumption we make about the structure of F is the following. If X and Y are variables and ϕ and ψ are members of F the following formulas should also be members of F.

$$X = Y \qquad\qquad \exists X.\phi \qquad\qquad \phi \wedge \psi$$

Given an assignment V, formulas ϕ and ψ, and variables X and Y, we expect the truth assignment $\models$ to satisfy the following.

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. $V \models \phi \wedge \psi$ iff $V \models \phi$ and $V \models \psi$

□

Note that the truth assignment for a formula $\phi \wedge \psi$ is uniquely determined by the truth assignments for the formulas ϕ and ψ. In the same way, if we know the truth assignment for ϕ, for assignments V, we can also determine the truth assignment for $\exists X.\phi$. It follows that we do not need to specify the truth assignments for conjunctions and existential quantifications in the definition of a pre-constraint system.

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

Let us consider a very simple example of a pre-constraint system.

Example 4.2.2 Let C be the set of natural numbers. Let the set of formulas F be the the smallest set that satisfies the axioms and contains the formula $X = n$, for each variable X and $n \in C$.

Say that $V \models X = n$ iff $V(X) = n$.

The truth assignment for other formulas can be derived from the axioms of pre-constraint systems. □

The general definition allows very powerful constraint systems, where the basic operations can be computationally expensive, or even uncomputable. If we want a concurrent language that can be implemented efficiently, we should of course choose a constraint system where the basic operations (adding a constraint to the store and entailment) have simple and efficient implementations. 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 4.2.3 Suppose we have a set of constant symbols $\{a, \ldots\}$ and function symbols $\{f, \ldots\}$. Define a set of *expressions* according to

1. a is an expression, for any constant symbol a.
2. $f(E_1, \ldots, E_n)$ is an expression, if f is an n-ary function symbol and $E_1, \ldots, E_n$ are expressions.
3. X is an expression, if X is a variable.

The formulas in F are simply formulas of the form $E_1 = E_2$, where E_1, E_2 are expressions. Let C be the set of expressions that do not contain variables.

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 4.2.4 A *constraint* is a non-empty set c of formulas, such that

1. if $\phi \in c$, and $\psi \preceq \phi$, then $\psi \in c$, and
2. if $\phi, \psi \in c$, then $\phi \wedge \psi \in c$.

□

For a formula ϕ let $[\phi] = \{\psi \mid \psi \preceq \phi\}$. Clearly $[\phi]$ is a 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.

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, $\{X = X, Y = Y, \ldots\}$. We will use the usual relation symbol $\sqsubseteq$ for inclusion between constraints, so that $c \sqsubseteq d$ if and only if $c \subseteq d$, and the symbol $\sqcup$ for least upper bound. Say that a constraint c is *finite* if whenever R is a directed set such that $c \sqsubseteq \bigsqcup R$, there is some $d \in R$ such that $c \sqsubseteq d$. Let $\mathcal{U}$ be the set of constraints, and $\mathcal{K}(\mathcal{U})$ the set of finite constraints. Note that for formulas ϕ and ψ, we have $[\phi] \sqcup [\psi] = [\phi \wedge \psi]$.

In the constraint programming language given in next section, we will assume all ask and tell constraints to be finite. Thus, it is worthwhile to take a closer look at the finite constraints.

Proposition 4.2.5 [29, Proposition 4.12 (ii)] The finite constraints are exactly those constraints that can be given in the form $[\phi]$, for some formula ϕ.

Proof. ($\supseteq$) Let ϕ be a formula. We will show that $[\phi]$ is finite. Let R be a directed set such that $\bigsqcup R \sqsupseteq [\phi]$. We have $\bigcup R \supseteq [\phi]$, and thus $\phi \in \bigcup R$. It follows immediately that $\phi \in c$, for some $c \in R$, and thus $[\phi] \sqsubseteq c$.

($\subseteq$) Let c be a finite constraint. Let $R = \{[\phi] \mid [\phi] \sqsubseteq c\}$. It is straightforward to establish that R is directed. We have $\bigsqcup R \sqsupseteq c$, since for any $\phi \in c$ we have $[\phi] \sqsubseteq c$, thus $[\phi] \in R$, and $\phi \in \bigsqcup R$. Since c is finite $\bigsqcup R \sqsupseteq c$ implies that $c \sqsubseteq d$ for some $d \in R$. We know that d is of the form $[\phi]$, for some ϕ such that $[\phi] \sqsubseteq c$. Thus $c = [\phi]$. □

As noted previously $[\phi] \sqcup [\psi] = [\phi \wedge \psi]$, for formulas ϕ and ψ. 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 an algebraic lattice.

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

1. $\exists_X([\phi]) = [\exists X.\phi]$, for formulas ϕ.

2. $\exists_X(\sqcup R) = \bigsqcup_{d \in R} \exists_X(d)$, for directed sets $R \subseteq \mathcal{K}(\mathcal{U})$.

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

For a constraint c and a variable X, say that c *depends on* X, if $\exists_X(c) \neq c$.

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 $X > 0$, $X > 1$, $X > 2$, ... hold, while $\exists X.X > n$ is true for any natural number n and thus the formulas

$$\exists X.X > 0, \quad \exists X.X > 1, \quad \exists X.X > 2, \quad \ldots$$

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, the resulting constraint system will contain the elements in the diagram below, in which the elements are totally ordered by $\sqsubseteq$.

$$\begin{array}{c} \top \\ \bigsqcup_{n \in \mathbf{N}}[X > n] \\ \cdot \\ \cdot \\ \cdot \\ [X > 2] \\ [X > 1] \\ [X > 0] \\ \bot \end{array}$$

□

4.2.2 Examples of constraint systems

We give some simple examples of constraint systems. Note that in the presentation of a constraint system, it is sufficient to give the domain of values, the set of formulas and the truth assignment. Since the formulas are always assumed to be closed under conjunction and contain the simple identities (equality between variables) we do not need to mention these formulas explicitly. Also, there is no need to specify the truth assignment for conjunctions and simple identities, since this is already given by the definition of pre-constraint systems.

Example 4.2.6 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. □

Example 4.2.7 (Rational intervals) Let the domain of values be the rational numbers, the formulas be of the form $X \in [r_1, r_2]$, where r_1 and r_2 are rational numbers. Let the truth assignment $\models$ be such that $V \models X \in [r_1, r_2]$ iff $r_1 \leq V(X) \leq r_2$. It should be clear that in this constraint system, entailment is computable. □

Example 4.2.8 (Real numbers) It is of course possible to construct constraint systems which can *not* not be implemented on a computer. Consider the following, which is based on the theory of the real numbers.

Let the domain of values be the real numbers, the formulas be of the form $X = E$, where E is some expression over the real numbers. Let the truth assignment $\models$ be such that $V \models X = E$, iff E evaluates to $V(X)$, where V gives the values to variables occurring in E. □

4.2.3 From formulas to constraints

When we prove things about the constraint system, it is convenient to be able to relate constraints and truth assignments.

Given an assignment V and a constraint c, 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 $V \models \phi$, for formulas ϕ.
2. $V \models \exists_X c$ iff $V' \models c$, for an assignment V' such that $V(Y) = V'(Y)$, for variables Y distinct from X.
3. $V \models c \cup d$ iff $V \models c$ and $V \models d$.

In the rest of this text, we will not make a syntactic disinction between formulas and finite constraints. For example, the finite constraint $[X = Y]$ will be written $X = Y$.

4.2.4 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 [30].

Proposition 4.2.9 Given a pre-constraint system $\langle F, Var, \models, C\rangle$, let the lattice $\langle \mathcal{U}, \sqsubseteq\rangle$ be the corresponding domain of constraints, with $\bot$ and $\top$

the least and greatest elements of $\mathcal{U}$. The following postulates are satisfied for any constraints $c, d \in \mathcal{U}$ and any variables $X, Y, Z \in \mathit{Var}$.

1. the structure $(\mathcal{U}, \sqsubseteq)$ forms an algebraic lattice.
2. $\exists_X$ is a continuous function $\exists_X : \mathcal{U} \to \mathcal{U}$,
3. $\exists_X(\top) = \top$,
4. $\exists_X(c) \sqsubseteq c$,
5. $\exists_X\,(c \sqcup \exists_X(d)) = \exists_X(c) \sqcup \exists_X(d)$,
6. $\exists_X(\exists_Y(c)) = \exists_Y(\exists_X(c))$,
7. $(X = X) = \bot$,
8. $(X = Y) = \exists_Z(X = Z \sqcup Z = Y)$, for Z distinct from X and Y,
9. $c \sqsubseteq (X = Y) \sqcup \exists_X(X = Y \sqcup c)$, for X and Y distinct.

Items 3-9 are borrowed from cylindric algebra. However, the structure is not necessarily a cylindric algebra, since a cylindric algebra is required to satisfy the axioms of Boolean algebra and must thus be a distributive lattice, while it is possible to construct a constraint system which is not distributive. For example, the constraint system derived from the pre-constraint system in Example 4.2.2 is not a distributive lattice, since it contains the sub-lattice $\{\mathbf{true}, X = 1, X = 2, X = 3, \mathbf{false}\}$.

4.2.5 Remarks

We have given a general framework for the construction of constraint systems. In the development of the semantic models in the following chapters, the properties that will be important for us are that the constraint system satisfies the axioms of cylindric algebra listed above, and that the constraints form an algebraic lattice. In contrast, Saraswat, Rinard and Panangaden [71] require in their semantics for non-deterministic ccp that the constraint system should be *finitary*, i.e., that for each finite constraint there should only be a finite set of smaller finite constraints. As an example of a constraint system that is *not* finitary, they mention the constraint system of rational intervals, as described in Example 4.2.7.

4.3 Syntax of ccp

In this section, we define a concurrent constraint programming language and give its operational semantics. However, the operational semantics, which is given as a set of computation rules is in itself not able to distinguish between computations that are fair and those that are not. To specify the set of fair computations we give an inductive definition of fairness.

This definition is, to the best of my knowledge, the first formal definition of fairness for a concurrent constraint programming language.

We assume a set $\mathcal{N}$ of procedure symbols $p, q, \ldots$. The syntax of an *agent* A is given as follows, where c ranges over finite constraints, and X over variables.

$$\begin{array}{rl} A ::= & c \mid \\ & \bigwedge_{j \in I} A^j \mid \\ & (c_1 \Rightarrow A_1 \,[]\, \ldots \,[]\, c_n \Rightarrow A_n) \mid \\ & \exists^c_X A \mid \\ & p(X) \end{array}$$

A tell constraint, written c, 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 countable, represents a parallel composition of the agents A^j. We will use $A_1 \wedge A_2$ as a shorthand for $\bigwedge_{j \in \{1,2\}} A^j$. In a conjunction, the indices are written as superscripts instead of subscripts to avoid confusion with subscripts representing positions in a computation (to be introduced in Definition 4.4.1). An agent $(c_1 \Rightarrow A_1 \,[]\, \ldots \,[]\, c_n \Rightarrow A_n)$ represents a selection. If one of the ask constraints c_i becomes true, the corresponding agent A_i may be executed. Agents of the form $\exists^c_X A$ represent agents with local data. The variable X is local, which means that the value of X is not visible to the outside. The constraint c is used to represent the constraint on the local value of X between computation steps.

Note that the syntax for agents describes both agents appearing in a program, and agents appearing as intermediate states in a computation. However, we will assume that agents of the form $\exists^c_X A$ occuring in a program or in the initial state of a computation will always have $c = \bot$. When this is the case, the local store can be omitted and the agent written $\exists_X A$.

A *program* Π is a set of definitions of the form $p(X) :: A$, where each procedure symbol p occurs in the left-hand side of exactly one definition in the program.

Remark To simplify the presentation, we only consider definitions with one argument. If we assume a suitable constraint system, such as the term model, we can use a function symbol (e.g. f) as a tuple constructor and let the formula $f(E_1, \ldots, E_n)$ represent a tuple of the n arguments. $\square$

4.4 Operational semantics

A *configuration* is a pair $A : c$ consisting of an agent A acting on a finite constraint c. The latter will be referred to as the *store* of the configuration. The operational semantics is given through a relation $\longrightarrow$ over configurations, assuming a program Π. For any computation step $A : c \longrightarrow A' : c'$, the constraint c' will always contain more information than c, i.e., $c \sqsubseteq c'$, so a computation step is never destructive.

We say that a variable X is *bound to* a value v if the current store c is such that for any variable assignment V such that $V \models c$, we have $V(X) = v$. Similarly, we say that we *bind* a variable to a value if by adding constraints to the store we make sure that the variable is bound to the value in the resulting store.

We present rules that define $\longrightarrow$ in the usual style of structural operational semantics [65].

1. The *tell constraint* simply adds new information (itself) to the environment.
$$c : d \longrightarrow c : c \sqcup d$$

2. A *conjunction* of agents is executed by interleaving the execution of its components.
$$\frac{A^k : c \longrightarrow B^k : d, \quad k \in I}{\bigwedge_{j \in I} A^j : c \longrightarrow \bigwedge_{j \in I} B^j : d} \quad ,$$
where $B^j = A^j$, for $j \in I \setminus \{k\}$.

3. If one of the ask constraints in a *selection* is satisfied by the current environment, the selection can be reduced to the corresponding agent.
$$\frac{c_i \sqsubseteq c}{(c_1 \Rightarrow A_1 \,[]\, \ldots \,[]\, c_n \Rightarrow A_n) : c \longrightarrow A_i : c}$$

4. A configuration with an existentially quantified agent $\exists_X^c A : d$ is executed one step by doing the following. Apply the function $\exists_X$ to the present environment (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 environment. A computation step is performed in the local environment, which gives a new local environment (c' say). To transmit any results to the global environment, the function $\exists_X$ is again applied to hide any information relating to the variable X. The

constraint thus obtained is combined with the previous global environment d. The local environment c' is stored as part of the existential quantification.

$$\frac{A : c \sqcup \exists_X(d) \longrightarrow A' : c'}{\exists^c_X A : d \longrightarrow \exists^{c'}_X A' : d \sqcup \exists_X(c')}$$

5. A *call* to a procedure is reduced to the body of its definition.

$$p(X) : c \longrightarrow A[X/Y] : c,$$

where the definition $p(Y) :: A$ is a member of Π, and the 'substitution' $A[X/Y]$ is a shorthand[1] for

$$\exists_\alpha(\alpha = X \wedge \exists_Y(\alpha = Y \wedge A)),$$

where α is a variable that does not occur in the program.

4.4.1 Some simple computation examples

We apply the computation rules to some simple agents.

First, a conjunction of a selection and a tell constraint. Consider the configuration

$$(X = 5 \Rightarrow Y = 7) \wedge X = 5 : \bot.$$

The selection cannot execute, since the ask constraint is not entailed by the store (that is, $\bot$). The tell constraint is executable, so we can perform the computation step

$$\begin{aligned}(X = 5 \Rightarrow Y = 7) \wedge X = 5 : \bot& \\ \longrightarrow (X = 5 \Rightarrow Y = 7) \wedge X = 5 : X = 5.&\end{aligned}$$

(The tell constraint still remains in the conjunction, even though it is now redundant.) Now as the constraint $X = 5$ has been added to the store, the selection can execute, as the ask constraint of its only alternative is entailed.

$$(X = 5 \Rightarrow Y = 7) \wedge X = 5 : X = 5 \longrightarrow Y = 7 \wedge X = 5 : X = 5$$

As we have replaced the selection by its only branch, we see immediately that another computation step is possible.

$$Y = 7 \wedge X = 5 : X = 5 \longrightarrow Y = 7 \wedge X = 5 : X = 5 \wedge Y = 7$$

Note that the constraint $X = 5 \wedge Y = 7$ is equivalent to $(X = 5) \sqcup (Y = 7)$. We chose the former notation since it is more readable.

[1]We could use $A[X/Y] \equiv \exists_Y(Y = X \wedge A)$, if we knew that the variables X and Y were always distinct.

Next we consider a computation which involves hidden data. The reader may find it helpful to take a look at the computation rules for existential quantifications before proceeding. Let A be the agent

$$(X = 5 \Rightarrow Y = 7) \wedge (Y = 7 \Rightarrow Z = 3)$$

and consider the configuration

$$\exists_Y A : \bot.$$

(Here, the local data of the existential quantification is $\bot$.) To perform a computation step by the existential quantification, we must ask ourselves if the configuration

$$A : \bot$$

can perform a computation step. Clearly, since the agent A consists of a conjunction of two selections, and neither of the tests (ask constraints) are entailed by the store, it follows that A can not perform any computation step. Suppose now that input arrives from the outside, and we find that the store contains the constraint $X = 5$. To perform a computation step with the configuration

$$\exists_Y A : X = 5,$$

we consider the 'local' configuration

$$A : X = 5,$$

where the store was obtained by $\bot \sqcup \exists_Y (X = 5) = (X = 5)$. We see that the test of the first selection of A is entailed by the store, so we can perform the computation step

$$A : X = 5 \longrightarrow Y = 7 \wedge (Y = 7 \Rightarrow Z = 3) : X = 5.$$

Thus, we have the computation step

$$\exists_Y A : X = 5 \longrightarrow \exists_Y^{X=5}(Y = 7 \wedge (Y = 7 \Rightarrow Z = 3)) : X = 5.$$

To see if the existential quantification can do another step, we look again at the local configuration

$$Y = 7 \wedge (Y = 7 \Rightarrow Z = 3) : X = 5.$$

We can perform a local step

$$\begin{aligned} &Y = 7 \wedge (Y = 7 \Rightarrow Z = 3) : X = 5 \\ &\qquad\qquad \longrightarrow Y = 7 \wedge (Y = 7 \Rightarrow Z = 3) : X = 5 \wedge Y = 7 \end{aligned}$$

which corresponds to the step

$$\begin{aligned}&\exists_Y^{X=5}(Y = 7 \wedge (Y = 7 \Rightarrow Z = 3)) : X = 5\\ &\qquad\longrightarrow \exists_Y^{X=5\wedge Y=7}(Y = 7 \wedge (Y = 7 \Rightarrow Z = 3)) : X = 5\end{aligned}$$

at the higher level. Since $\exists_Y(X = 5 \wedge Y = 7)$ is equal to $(X = 5)$, the constraint concerning the value of Y is not visible outside the quantification. However, the local value of Y is recorded in the local store. Last two steps are straight-forward. We do a local step

$$\begin{aligned}&Y = 7 \wedge (Y = 7 \Rightarrow Z = 3) : X = 5\\ &\qquad\longrightarrow Y = 7 \wedge Z = 3 : X = 5 \wedge Y = 7,\end{aligned}$$

corresponding to the global step

$$\begin{aligned}&\exists_Y^{X=5\wedge Y=7}(Y = 7 \wedge (Y = 7 \Rightarrow Z = 3)) : X = 5\\ &\qquad\longrightarrow \exists_Y^{X=5\wedge Y=7}(Y = 7 \wedge Z = 3) : X = 5,\end{aligned}$$

and finally the local step

$$\begin{aligned}&Y = 7 \wedge Z = 3 : X = 5 \wedge Y = 7\\ &\qquad\longrightarrow Y = 7 \wedge Z = 3 : X = 5 \wedge Y = 7 \wedge Z = 3,\end{aligned}$$

which corresponds to the global step

$$\begin{aligned}&\exists_Y^{X=5\wedge Y=7}(Y = 7 \wedge Z = 3) : X = 5\\ &\longrightarrow \exists_Y^{X=5\wedge Y=7\wedge Z=3}(Y = 7 \wedge Z = 3) : X = 5 \wedge Z = 3.\end{aligned}$$

Thus, we reach a configuration in which the store is $X = 5 \wedge Z = 3$ and no other configurations can be reached by computation steps.

4.4.2 Computations

Using the operational definition we can specify the set of computations. The basic idea is that in a computation the store can either be modified by the agent, during a computation step, or by the outside, during an input step.

Definition 4.4.1 Assuming a program Π, a *computation* is an infinite sequence of configurations $(A_i : c_i)_{i\in\omega}$ such that for all $i \geq 0$, we have either $A_i : c_i \longrightarrow A_{i+1} : c_{i+1}$ (*a computation step*), or $A_i = A_{i+1}$ and $c_i \sqsubseteq c_{i+1}$ (*an input step*). □

Note that some steps are both computation steps and input steps. For example, going from

$$X = 5 : \bot \quad \text{to} \quad X = 5 : X = 5$$

can be done either in a computation step, or in an input step.

In the following text, we will leave out references to the program Π when we can do so without causing ambiguities.

An input step from $A : c$ to $A : c'$, such that $c = c'$ is an *empty input step*. A computation where all input steps are empty is a *non-interactive computation*.

Remark According to the definition above, all computations are infinite. However, since one can see a finite computation as an infinite computation which ends in an infinite sequence of empty input steps, we do not lose in generality by only considering infinite computations. $\square$

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? A direct formalisation of the intuitive fairness requirement would fail to 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.

How should we specify the set of fair computations? First, note that a computation can often be considered to contain other computations. For example, to perform a computation step with a process $A \wedge B : c$, 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 4.5.1 Let the relation *immediate inner computation of* be the weakest relation over ω-sequences of configurations which satisfies the following.

1. $(A_i^k : c_i)_{i\in\omega}$ is an immediate inner computation of $(\bigwedge_{j\in I} A_i^j : c_i)_{i\in\omega}$, for $k \in I$.

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

The relation '*inner computation of*' is defined to be the transitive and reflexive closure over the relation 'immediate inner computation of'. □

Now, we would expect the inner computations of a computation to also be computations.

Proposition 4.5.2 If $(A_i : c_i)_{i\in\omega}$ is computation, and $(B_i : d_i)_{i\in\omega}$ is an inner computation of $(A_i : c_i)_{i\in\omega}$, then $(B_i : d_i)_{i\in\omega}$ is also a computation.

Proof. We first consider the case when $(B_i : d_i)_{i\in\omega}$ is an immediate inner computation of $(A_i : c_i)_{i\in\omega}$. Suppose $(A_i : c_i)_{i\in\omega}$ is a computation. Then $(A_i : c_i)_{i\in\omega}$ is in one of the two forms given by the definition of the relation 'immediate inner computation'.

If for all $i \geq 0$, $A_i = \bigwedge_{j\in I} A_i^j$, then we must have $B_i = A_i^k$ for all $i \geq 0$, and some $k \in I$, and $c_i = d_i$ for all $i \geq 0$. So for each $i \geq 0$, we have either $B_i = B_{i+1}$, or $B_i : d_i \longrightarrow B_{i+1} : d_{i+1}$.

Suppose that for all $i \geq 0$, $A_i = \exists^{e_i}_X B_i$ where $d_i = \exists_X(c_i) \sqcup e_i$. Consider a fixed i. If the ith step of $(A_i : c_i)_{i\in\omega}$ is a reduction step, it follows, by the computation rules, that $B_i : \exists_X(c_i) \sqcup e_i \longrightarrow B_{i+1} : e_{i+1}$. It remains to be proved that $e_{i+1} = \exists_X(c_{i+1}) \sqcup e_{i+1}$. It is sufficient to show that $e_{i+1} \sqsupseteq \exists_X(c_{i+1})$. By the reduction rule, $c_{i+1} = \exists_X(e_{i+1}) \sqcup c_i$. By the properties of the constraint system, $\exists_X(c_{i+1}) = \exists_X(e_{i+1}) \sqcup \exists_X(c_i)$. Since $e_{i+1} \sqsupseteq \exists_X(c_i)$, and $e_{i+1} \sqsupseteq \exists_X(e_{i+1})$, we see that $e_{i+1} \sqsupseteq \exists_X(c_{i+1})$, and we have established that the ith step of $(B_i : d_i)_{i\in\omega}$ is a reduction step.

If the ith step of $(A_i : c_i)_{i\in\omega}$ is an input step, then the ith step of $(B_i : d_i)_{i\in\omega}$ is also an input step, by monotonicity of $\sqcup$ and $\exists_X$.

The general case can be shown by induction on the nesting depth. □

We will define the fairness requirement in a bottom-up fashion by giving a sequence of auxiliary definitions which capture different aspects of fairness. First, the weakest and simplest fairness property, top-level fairness. A computation is top-level fair unless the first agent of the computation is a tell constraint that is never added to the store, or a selection which has an alternative that can be selected, but no alternative is selected, or a call that is never reduced to its definition. Using top-level fairness we can specify initial fairness, which concerns agents occurring as a part of the first agent, and finally the actual definition of fairness.

Definition 4.5.3 A computation $(A_i : c_i)_{i\in\omega}$ 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 $c_i \sqsupseteq c$.

3. If $A_0 = (d_1 \Rightarrow B_1 [] \ldots [] d_n \Rightarrow B_n)$, and $d_j \sqsubseteq c_0$ for some $j \leq n$, then there is an $i \geq 0$ such that $A_i \neq A_0$.

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. □

4.5.1 Informal justification of the definition of fairness

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. We argue that if a computation is fair in the formal sense if and only if it is fair in the intuitive sense.

Suppose that we have a computation x, which is fair in the formal sense. Consider an agent A which occurs somewhere in the computation. Consider the suffix x' of the computation, which is selected so that the agent A occurs in the first configuration of x'. By the definition of fairness, the computation x' is initially fair. This means that every inner computation of x' 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, since fairness requires that the store should eventually entail c. 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, we 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, 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 x 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 x, clearly any suffix x' of x is also fair in the intuitive sense. We now want to show that each inner computation of the suffix x' is top-level fair. Suppose A occurs in the first configuration of x' (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 that 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, i.e., be replaced by the body of the corresponding definition. If A is a selection in which one of the conditions is entailed, it follows that A is able to perform a computation step and thus will eventually do so. If A is a selection in which none of the conditions is entailed, it follows immediately that the computation is top-level fair.

It follows that the inner computation beginning with A is top-level fair. It follows that every inner computation of x' is top-level fair, thus x' is initially fair. We draw the conclusion that every suffix of x is initially fair, and it follows that the computation x 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 should be intuitively clear.

Proposition 4.5.4 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 $c_j \sqsupseteq c$.

The case when $A_0 = (d_1 \Rightarrow B_1 \mathbin{[]} \ldots \mathbin{[]} d_n \Rightarrow B_n)$ is similar. □

Proposition 4.5.5 If one suffix of a computation is initially fair, then the computation is initially fair.

Proof. Suppose we have a computation x where the kth suffix is initially fair. Consider an inner computation y of x. The kth suffix of y is an inner computation of the kth suffix of x and thus top-level fair. So by proposition 4.5.4 every inner computation of x is top-level fair and therefore x is initially fair. □

Lemma 4.5.6 If one suffix of a computation is 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 4.5.7 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 kth suffix of y is an inner computation of the kth suffix of x, which is initially fair. So the kth suffix of y and all its all its inner computations are top-level fair, which implies that the kth suffix of y is initially fair. So each suffix of y is initially fair, and thus is y fair. □

Lemma 4.5.8 A computation whose immediate inner computations are fair, and all suffixes are top-level fair, is fair.

(The second condition cannot be omitted; some computations do not have any immediate inner computations.)
Proof. First we show that a computation that satisfies the above is initially fair. Consider an arbitrary inner computation x. This inner computation x is either the computation itself, which is top-level fair, or an inner computation of one of the immediate inner computations, from which follows that x is top-level fair, since it is the inner computation of a computation which is fair and thus also initially fair.

Now, consider the kth suffix of the computation. This suffix satisfies the conditions stated in the lemma; all its immediate inner computations are fair and all its suffixes are top-level fair. Thus the kth suffix is initially fair (by the reasoning in the previous paragraph), and thus the computation is fair. □

4.6 Closure operators and Deterministic Programs

In this section we will review some results from lattice theory concerning closure operators, and how closure operators can be used to give the semantics of certain concurrent constraint programs.

A *deterministic ccp program* is a ccp program in which all selections have only one alternative. Since non-determinism in ccp stems from selections in which on some occasions more than one alternative can be selected, the restriction to selections with only one alternative effectively makes the programs deterministic. This is perhaps not completely obvious, since agents still execute concurrently, and results may be computed in different orders. However, as shown by Saraswat, Rinard, and Panangaden [71], it turns out that the agents may be represented as functions with some special properties, and that the meaning of programs may be obtained as simple fixpoints, as in Kahn's semantics [37].

The functions which represent deterministic ccp programs are called closure operators. The definition of closure operators and some of their properties will be given in the following part of this section. In the final part of this section we will briefly review the semantics of determinate ccp.

4.6.1 Closure operators

Jagadeesan, Panangaden and Pingali [33] showed how a concurrent process operating over a domain that allows 'logic variables', i.e., 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 [71]. This section gives the definition of closure operators and some of their properties. See reference [29] for further results on closure operators.

Let us look at an agent as a function f that takes a store as input, and returns a new store. What properties are satisfied by the agent? First, the agent may never remove anything from the store, so the resulting store is always stronger than the original store. Thus, we have $f(x) \geq x$, for all x. Second, we assume that the process has finished all it wanted to do when it returned, thus applying it again will not change anything. Thus we have $f(f(x)) = f(x)$, for all x. Putting these two points together gives us the definition of closure operators.

Definition 4.6.1 For an algebraic lattice $(D, \sqsubseteq)$, a *closure operator* over D is a monotone function f over D with the property that $f(x) \sqsupseteq x$ 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 an algebraic lattice D is the set $f(D) = \{f(x) \mid x \in D\}$. 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, $\sqcap T \in S$. This is easy to see if we observe that $f(\sqcap T) \sqsubseteq \sqcap\{f(x) \mid x \in T\}$, since f is monotone, $\sqcap T \sqsubseteq f(\sqcap T)$, since f is a closure operator, and $\sqcap\{f(x) \mid x \in T\} = \sqcap T$, 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 $f_S(x) = \sqcap(\{x\}^u \cap S)$, 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 $\sqcap$. In the subsequent text we will take advantage of this property, and sometimes see closure operators as functions, and sometimes as sets. 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 an algebraic lattice form a complete lattice. Consider the functions over an algebraic lattice to be ordered point-wise, i.e., $f \sqsubseteq g$ iff $f(x) \sqsubseteq g(x)$ for all x. Now we have $f \sqsubseteq g$ if and only if $f \supseteq g$. If $\{f_i\}_{i \in I}$ is a family of closure operators, it is easy to see that $\bigcap_{i \in I} f_i$ 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 an algebraic lattice D is the function that maps every element of the algebraic lattice to $\top$, and the bottom element is the identity function.

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

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

Since a closure operator is characterised by its set of fixpoints, 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 $\{y\}^u = \{z \in D \mid y \sqsubseteq z\}$. 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.

4.6.2 Semantics of deterministic ccp

In this section we briefly review the results of Saraswat, Rinard, and Panangaden [71] concerning the semantics of deterministic ccp.

The idea is that we should try to model each agent as a closure operator that takes a store as input and returns a new store.

First, consider a tell constraint c. Applying c to a store d gives us the store $c \sqcup d$. Thus the tell constraint c can be modelled with the closure operator $(\bot \to c)$.

To model a selection $(c \Rightarrow A)$, let us assume that the agent A can be modelled with the closure operator f. When examining the behaviour of the selection, we see that it remains passive until the ask constraint c is entailed by the store, and then the selection behaves exactly like the agent A. Thus we can model the selection with the closure operator $(c \to f)$, which simply returns its input when c is not less that or equal to the input, and applies f to the input when the input is stronger than c.

To model a conjunction $A \wedge B$ (we only consider the finite conjunction here, generalisation is straight-forward), we assume that the semantics of A is the closure operator f, and that the semantics of the agent B is the closure operator g. Given a store c, the result of running the agent A is the new store $f(c)$. If we now run the agent B we obtain a new store $g(f(c))$. But now A can execute further and produce the store $f(g(f(c)))$. This can go on forever.

Suppose now that A and B are allowed to interleave forever, thus producing the limit of the sequence of stores indicated above, what will the limit look like? It is easy to show by a mathematical argument (assuming that f and g are continuous) that the limit must be a fixpoint of both f and g. Also, it can be shown that the limit must be the smallest mutual fixpoint of f and g greater than c. Thus, if we want a function that models the behaviour of $A \wedge B$, we should use the least closure operator stronger than f and g, which is $f \cap g$.

Next, consider an existential quantification $\exists_X A$, where the semantics of A is given by the closure operator f. If we run the agent $\exists_X A$ with a store c, the store accessible to A is given by $\exists_X(c)$. If the agent A produces a new store d, we see that the part of the modification visible on the outside is $\exists_X(d)$. For example, if c entails the constraint $X = 10$, this aspect of c is not visible to A. If A chooses to add the tell constraint $X = 5$ to the store, this change is not visible to an outside observer.

Thus, we have a form of two-way hiding, and the semantics of $\exists_X A$ can be given by the closure operator g, given as follows.

$$g(c) = c \sqcup \exists_X(f(\exists_X(c)))$$

To deal with the general case, we define a function E_X, which takes a function and returns the corresponding function where X is hidden. E_X can be defined as follows.

$$\mathrm{E}_X(f) = \mathbf{id} \sqcup (\exists_X \circ f \circ \exists_X).$$

Now, if the semantics of A is f, the semantics of $\exists_X A$ is $\mathrm{E}_X(f)$.

We have seen that the basic constructs of determinate ccp can be modelled as continuous closure operators. It is time to write down the properties of the constructs in the form of a fixpoint semantics. We will give a fixpoint semantics of deterministic ccp in which each agent is mapped to a continuous closure operator, and a program is mapped to a function from names to closure operators, i.e., an *environment*. Thus, the semantics for

Definition of $\mathcal{E}[\![A]\!] : (\mathcal{U} \to \mathcal{U})^{\mathcal{N}} \to (\mathcal{U} \to \mathcal{U})$

$$\begin{aligned}
\mathcal{E}[\![c]\!]\sigma &= (\perp \to c) \\
\mathcal{E}[\![\textstyle\bigwedge_{j \in I} A^j]\!]\sigma &= \bigcap_{j \in I} \mathcal{E}[\![A^j]\!]\sigma \\
\mathcal{E}[\![(c \Rightarrow A)]\!]\sigma &= (c \to \mathcal{E}[\![A]\!]\sigma) \\
\mathcal{E}[\![\exists_X A]\!]\sigma &= \mathrm{E}_X(\mathcal{E}[\![A]\!]\sigma) \\
\mathcal{E}[\![p(X)]\!]\sigma &= \mathrm{E}_\alpha((\alpha = X) \cap (\sigma p))
\end{aligned}$$

Definition of $\mathcal{P}[\![\Pi]\!] : (\mathcal{U} \to \mathcal{U})^{\mathcal{N}} \to (\mathcal{U} \to \mathcal{U})^{\mathcal{N}}$

$$\mathcal{P}[\![\Pi]\!]\sigma p = \mathrm{E}_Y((Y = \alpha) \cap (\mathcal{E}[\![A]\!]\sigma)),$$

where for each $p \in \mathcal{N}$ the definition in Π is assumed to be of the form $p(Y) :: A$, for some variable Y and some agent A

Figure 4.1: The fixpoint semantics for determinate ccp

an agent A is given as a function $\mathcal{E}[\![A]\!] : (\mathcal{U} \to \mathcal{U})^{\mathcal{N}} \to (\mathcal{U} \to \mathcal{U})$, which maps environments to closure operators.

Now, to give the semantics of a call $p(X)$, we model the substitution using hiding together with equality, so that the dummy variable α is used for argument-passing, just like in the operational semantics. Thus the semantic rule for procedure calls becomes

$$\mathcal{E}[\![p(X)]\!]\sigma = \mathrm{E}_\alpha((\alpha = X) \cap (\sigma p)).$$

In the same way, to give the semantics of the procedure definitions in a program Π, we define a function $\mathcal{P}[\![\Pi]\!]$ which takes a program and an environment and produces a 'better' environment, as below. Assume that for each name p, the corresponding definition in Π is $p(Y) :: A$.

$$\mathcal{P}[\![\Pi]\!]\sigma p = \mathrm{E}_Y((Y = \alpha) \cap (\mathcal{E}[\![A]\!]\sigma))$$

The semantics of a program Π is the least fixpoint of $\mathcal{P}[\![\Pi]\!]$. We can now put the fixpoint semantics together, as shown in Figure 4.1.

4.7 Result and Trace Semantics

Turning back to the general problem of giving semantics for potentially non-deterministic concurrent constraint programs, we present two sematics based directly on the operational model of concurrent constraint programming presented in the earlier chapters.

We first define 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.

4.7.1 Result Semantics

Consider the situation where we run an agent with no interaction with other agents. If the agent terminates, we say that the result of the computation is the final contents of the store. If the agent does not terminate, we record the limit of the successive stores of the computation and say that the limit is the result of the computation. So, the *result semantics* for a given agent is a function from initial stores to the results of all possible computations.

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

$$\begin{aligned}\mathcal{R}_\Pi[\![A]\!]c = \{\textstyle\bigsqcup_{i\in\omega} c_i \mid (A_i : c_i)_{i\in\omega} &\text{ is a fair} \\ &\text{non-interactive computation with } A_0 : c_0 = A : c.\}\end{aligned}$$

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.

4.7.2 Traces

Remember that a computation is defined to be a sequence of configurations $(A_i : c_i)_{i\in\omega}$, where the environments, that is, the c_i's, are the only part of the computation visible to the outside. Now, a computation can go from $A_i : c_i$ to $A_{i+1} : c_{i+1}$ either by performing a computation step, or by receiving input, and this distinction is of course relevant when comparing behaviours of different agents.

In the trace semantics, an agent is represented by a set of traces, where 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.

Definition 4.7.1 A *trace* t is a pair $t = (v(t), r(t))$, where $v(t)$ is an ω-chain in $\mathcal{K}(\mathcal{U})$ and $r(t) \subseteq \omega$. The set of traces is denoted TRACE. □

The *trace of a computation* $(A_i : c_i)_{i \in \omega}$ is a trace $t = ((c_i)_{i \in \omega}, r)$, where the step from $A_i : c_i$ to $A_{i+1} : c_{i+1}$ is a computation step when $i \in r$, and an input step when $i \notin r$. We will sometimes use the notation $v(t)_i$ to refer to the ith element of the store sequence of t.

The trace semantics of an agent A, assuming a program Π, is defined as follows.

Definition 4.7.2 The function $\mathcal{O}_\Pi : \text{AGENT} \to \wp(\text{TRACE})$ is defined so that $t \in \mathcal{O}_\Pi[\![A]\!]$, iff t is the trace of a fair computation $(A_i : c_i)_{i \in \omega}$, where $A_0 = A$.

When the above holds, we say that the computation $(A_i : c_i)_{i \in \omega}$ *connects* the trace t to the agent. □

Operational semantics of simple agents

The operational semantics of tell constraints and calls can be given directly.

Proposition 4.7.3 For a tell constraint c, we have $t \in \mathcal{O}_\Pi[\![c]\!]$ iff

1. $v(t)_i \sqsupseteq c$, for some $i \in \omega$ and

2. $v(t)_{i+1} = v(t)_i \sqcup c$, for all $i \in r(t)$.

It is easy to see that in each computation step $c : d_i \longrightarrow c : d_{i+1}$, we have $d_{i+1} = d_i \sqcup c$. Fairness guarantees that the limit of the trace will be stronger than c.

Proposition 4.7.4 For a call $p(X)$, we have $\mathcal{O}_\Pi[\![p(X)]\!] = \mathcal{O}_\Pi[\![A[X/Y]]\!]$ where the definition of p in the program Π is $p(Y) :: A$.

4.7.3 Compositionality

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

Proposition 4.7.5 Assume an agent $\bigwedge_{j\in I} A^j$. For a trace t we have $t \in \mathcal{O}_\Pi[\![\bigwedge_{j\in I} A^j]\!]$ iff there are $t_j \in \mathcal{O}_\Pi[\![A^j]\!]$ for $j \in I$, such that $v(t_j) = v(t)$ for $j \in I$, $r(t) = \bigcup_{j\in I} r(t_j)$, and $r(t_i) \cap r(t_j) = \emptyset$, for $i, j \in I$ such that $i \neq j$.

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

That for all $j \in I$, each computation $(A_i^j : c_i)_{i\in\omega}$ is fair follows from Lemma 4.5.7 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 $v(t_j) = v(t)$, and $r(t_j) = \{i \in r(t) \mid k_i = j\}$. It is easy to check that the family of traces $\{t_j\}_{j\in I}$ satisfies the right-hand side of the proposition.

($\Leftarrow$) We know that for each $j \in I$ there is a fair computation $(A_i^j : c_i)_{i\in\omega}$ 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 $i \notin r(t_j)$, for $j \in I \setminus \{k_i\}$. By the computation rules it follows that

$$\bigwedge_{j\in I} A_i^j : c_i \longrightarrow \bigwedge_{j\in I} A_{i+1}^j : c_{i+1},$$

for all $i \in r(t)$. In the case that $i \in \omega \setminus r(t)$, it follows that $i \in \omega \setminus r(t_j)$, for all $j \in I$, and thus all computations $(A_i^j : c_i)_{i\in\omega}$ perform input steps at position i, which implies that $\bigwedge_{j\in I} A^j = \bigwedge_{j\in I} A_{i+1}^j$, from which follows that we can construct a computation $(\bigwedge_{j\in I} A_i^j : c_i)_{i\in\omega}$. Fairness follows from the fact that all immediate inner computations of the constructed computation are fair. $\square$

Proposition 4.7.6 Suppose we have an agent $\exists_X A$. For any trace t, we have $t \in \mathcal{O}_\Pi[\![\exists_X A]\!]$ iff there is a trace $u \in \mathcal{O}_\Pi[\![A]\!]$ such that with $v(t) = (d_i)_{i\in\omega}$ and $v(u) = (e_i)_{i\in\omega}$, we have

1. $r(t) = r(u)$
2. $e_0 = \exists_X(d_0)$,
3. for $i \in r(t)$, $d_{i+1} = d_i \sqcup \exists_X(e_{i+1})$, and

4. for $i \in \omega \setminus r(t)$, $e_{i+1} = e_i \sqcup \exists_X(d_{i+1})$.

Proof. ($\Rightarrow$) Suppose $t \in \mathcal{O}_\Pi[\![\exists_X A]\!]$. There is a computation $(\exists_X^{c_i} A_i : d_i)_{i\in\omega}$ that connects the trace t to the agent A. The computation has an inner computation $(A_i : c_i \sqcup \exists_X(d_i))_{i\in\omega}$ that we know, by Propositions 4.5.2 and 4.5.7, to be a fair computation. It remains to prove that with $e_i = c_i \sqcup \exists_X(d_i)$ Conditions 1-4 are satisfied.

Conditions 1 and 2 follow immediately (remember that $\exists_X A$ is short for $\exists_X^\perp A$). When $i \in r(t)$, it follows that $\exists_X^{c_i} A_i : d_i \longrightarrow \exists_X^{c_{i+1}} A_{i+1} : d_{i+1}$ and by the computation rules that $A_i : c_i \sqcup \exists_X(d_i) \longrightarrow c_{i+1}$, and $d_{i+1} = d_i \sqcup \exists_X(c_{i+1})$. By the properties of the constraint system we have $d_{i+1} = d_{i+1} \sqcup \exists_X(d_{i+1}) = d_i \sqcup \exists_X(c_{i+1}) \sqcup \exists_X(d_{i+1}) = d_i \sqcup \exists_X(c_{i+1} \sqcup \exists_X(d_{i+1})) = d_i \sqcup \exists_X(e_{i+1})$. Condition 3 follows immediately.

If $i \in \omega \setminus r(t)$ the corresponding step in the computation $(\exists_X^{c_i} A_i : d_i)_{i\in\omega}$ is an input step. This implies that $c_i = c_{i+1}$. So $e_{i+1} = c_{i+1} \sqcup \exists_X(d_{i+1}) = c_i \sqcup \exists_X(d_{i+1}) = c_i \sqcup \exists_X(d_i) \sqcup \exists_X(d_{i+1}) = e_i \sqcup \exists_X(d_{i+1})$.

($\Leftarrow$) Assume that the right-hand side of the proposition holds. There is a computation $(A_i : e_i)_{i\in\omega}$ that connects the trace u to the agent A. For all $i \in \omega$, let $c_i = \bigsqcup\{e_{j+1} \mid j < i,\ j \in r(t)\}$ (this should agree with the idea that c_i, which is the local data of the agent, only changes when the agent performs computation steps).

We want to show that $(\exists_X^{c_i} A_i : d_i)_{i\in\omega}$ is a fair computation that connects the trace t to the agent $\exists_X A$. Note that for all $i \in \omega$, it follows from our assumptions that $\exists_X(d_i) = \exists_X(e_i)$ and $c_i \sqcup \exists_X(d_i) = e_i$. If $i \in r(t) = r(u)$, we know that $c_{i+1} = e_{i+1}$ and $A_i : e_i \longrightarrow A_{i+1} : e_{i+1}$. By the computation rules and the equalities above,

$$\exists_X^{c_i} A_i : d_i \longrightarrow \exists_X^{c_{i+1}} A_{i+1} : d_{i+1}.$$

If $i \in \omega \setminus r(t)$ we have $A_{i+1} = A_i$ and $c_{i+1} = c_i$ so the ith step of $(\exists_X^{c_i} A_i : d_i)_{i\in\omega}$ is an input step.

To establish fairness of the computation $(\exists_X^{c_i} A_i : d_i)_{i\in\omega}$ it suffices to observe that its only immediate inner computation is fair. □

Proposition 4.7.7 $t \in \mathcal{O}_\Pi[\![(c_1 \Rightarrow A_1 \,[], \ldots [] \, c_n \Rightarrow A_n)]\!]$ iff one of the following holds.

1. $c_j \sqsubseteq v(t)_k$ for some $j \le n$ and $k \ge 0$, and there is a $u \in \mathcal{O}_\Pi[\![A_j]\!]$ such that for all $i \ge 0$, $v(u)_i = v(t)_{i+k+1}$, $v(t)_k = v(t)_{k+1}$, and $r(t) = \{i + k + 1 \mid i \in r(u)\} \cup \{k\}$.

2. There is no $j \le n$ and $k \ge 0$ such that $c_j \sqsubseteq v(t)_k$, and $r(t) = \emptyset$.

Proof. ($\Rightarrow$) Suppose $t \in \mathcal{O}_\Pi[\![(c_1 \Rightarrow A_1 \,[], \ldots [] \, c_n \Rightarrow A_n)]\!]$. Let $(B_i : d_i)_{i\in\omega}$ be the corresponding computation.

If $d_i \sqsupseteq c_l$, for some $i \in \omega$ and $l \leq n$, it follows by the fairness requirement that $B_{k+1} = A_j$ for some $k \geq 0$ and $j \leq n$. Since $(B_i : d_i)_{i \geq k+1}$ is a fair computation it follows that we have a corresponding trace $u \in \mathcal{O}_\Pi[\![A_j]\!]$. It is easy to see that the relationship between t and u are as stated in Condition 1.

If there is no $j \leq n$ and $i \in \omega$ such that $c_j \sqsubseteq d_i$, Condition 2 follows immediately.

($\Leftarrow$) Suppose Condition 1 holds. We will construct a fair computation $(B_i : d_i)_{i \in \omega}$ corresponding to the trace t. Let $(B_i : d_i)_{i \geq k+1}$ be the computation corresponding to the trace u. Let

$$B_0 = B_1 = \ldots = B_k = (c_1 \Rightarrow A_1 \, [], \ldots [] \, c_n \Rightarrow A_n)\,,$$

and

$$d_0 = v(t)_0,\ d_1 = v(t)_1, \ldots, d_k = v(t)_k.$$

It is straight-forward to check that $(B_i : d_i)_{i \in \omega}$ is a computation, and that t is the corresponding trace. Fairness follows from the fact that a suffix is known to be fair.

In the case when Condition 2 holds, it is easy to check that we can construct a completely passive computation of the selection which has trace t. □

Chapter 5

A Fully Abstract Semantics for ccp

In this chapter, we address the problem of developing a compositional and fully abstract semantics for concurrent constraint programming. A semantics is called fully abstract if it identifies exactly those programs that behave in the same way in any context. Thus full abstraction provides an optimal abstraction from internal details of the behaviour of a program, while preserving compositionality. We give a semantics which is fully abstract with respect to the result semantics.

Intuitively, a trace of a program can be obtained from a computation of the program by extracting the sequence of communication actions performed during the computation. In the case of data-flow networks, a communication action is the reception or transmission of a data item on a channel; for shared-variable programs, a communication action is an atomic change to the global shared state. In concurrent constraint programming, it seems natural to regard a communication action as the addition of information to the store.

In a concurrent constraint programming language, the set of all traces of a program gives a complete description of the behaviour of the program in all possible contexts, but it contains too much detail, i.e., it is not fully abstract. We solve this problem by adding an operation that forms the downward closure of the set of traces with respect to a partial order. Intuitively, this partial order captures the notion that one trace contains less information than another. We then show that the semantics obtained by applying this closure operation to the trace semantics is compositional and fully abstract with respect to the result semantics.

5.1 Related Work

A similar closure operation on traces has also been presented by Saraswat, Rinard, and Panangaden [71] but that work only considers finite behaviour. In contrast our semantics handles infinite computations and the associated notion of fairness, and can be seen as a natural extension of [71] to the infinite case.

See Section 2.8 for other results concerning fully abstract semantics of concurrent constraint programming.

5.2 Defining the fully abstract semantics

The fully abstract semantics is based on the idea 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 environments of the trace, that is,

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

The functionality of a trace can loosely be described as the function computed by an agent in one particular computation.

Definition 5.2.1 The *functionality* of a trace t, denoted $\mathrm{fn}(t)$, is the closure operator given by the following equation.

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

□

Note that this closure operator is the least closure operator f such that

$$v(t)_{i+1} \sqsubseteq f(v(t)_i),$$

for all $i \in r(t)$.

The following proposition offers a simple characterisation of $\mathrm{fn}(t)$ in terms of its fixpoints.

Proposition 5.2.2 Let t be a trace. A constraint d is a fixpoint of $\mathrm{fn}(t)$ exactly when for all $i \in r(t)$, $d \sqsupseteq (v(t)_i)$ implies $d \sqsupseteq (v(t)_{i+1})$.

Definition 5.2.3 We say that a trace t is a *subtrace* of a trace t', if the limit of t is equal to the limit of t' and the functionality of t is weaker than or equal to the functionality of t', i.e., $\mathrm{fn}(t) \supseteq \mathrm{fn}(t')$. A set $S \subseteq$ TRACE is *subtrace-closed*, if $t \in S$ whenever t is a subtrace of t' and $t' \in S$.

Given a trace t, the *inverse* of t is the trace $\bar{t} = (v(t), \omega \setminus r(t))$. □

5.2.1 Definition of the abstract semantics.

We can now define the abstract semantics, which we will prove to be fully abstract and compositional.

Definition 5.2.4 For an agent A, and a program Π, let

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

□

Not surprisingly, the abstract semantics contains sufficient information to allow the result semantics to be obtained from the abstract semantics. This is expressed in the following proposition.

Proposition 5.2.5 For an agent A, and constraint d, we have

$$\mathcal{R}_\Pi[\![A]\!]d = \{\lim(t) \mid t \in \mathcal{A}_\Pi[\![A]\!] \text{ and } \mathrm{fn}(t) = (d \to \lim(t))\}.$$

Example 5.2.6 As an example of the abstract semantics, consider the following two agents. Let the agent A_1 be

$$X = [1, 2, 3, 4],$$

and the agent A_2 be

$$\exists_Y (X = [1, 2 \mid Y] \wedge Y = [3, 4]).$$

The two agents produce the same result, and there is no way a concurrently executing agent could see that the agent A_2 produces the list in two steps, so we would expect these two agents to have the same abstract semantics.

A typical trace of A_1 might be the trace t_1, where

$$\begin{array}{rlll} v(t_1) = (\bot, & X = [1, 2, 3, 4], & \ldots) \\ r(t_1) = \{0 & & \}, \end{array}$$

and a typical trace of A_2 might be the trace t_2, where

$$\begin{array}{rllll} v(t_2) = (\bot, & \exists_Y (X = [1, 2 \mid Y]), & X = [1, 2, 3, 4], & \ldots) \\ r(t_2) = \{0, & 1 & & \}. \end{array}$$

We also see that $\mathrm{fn}(t_1) = \mathrm{fn}(t_2) = (\bot \to X = [1, 2, 3, 4])$, and $\lim(t_1) = \lim(t_2) = (X = [1, 2, 3, 4])$. The two traces have the same functionality and limit and are thus subtraces of each other. It is easy to see that any trace of A_1 is a subtrace of some trace of A_2, and vice verca, so it follows that the two agents have the same abstract semantics. □

5.2.2 Relationship with determinate semantics

Recall that for deterministic ccp programs there is a simple fully abstract fixpoint semantics where the semantics of an agent is given as a closure operator (Section 4.6). Given a deterministic agent A and program Π, where the semantics of A is given by the closure operator f, what does the corresponding abstract semantics for A look like?

For finite constraints c and d, if $f(c) \sqsupseteq d$ it follows that A will, given a store where c holds, add constraints to the store so that d is entailed. If, on the other hand, $f(c) \not\sqsupseteq d$, we can conclude that if A starts executing with the store c, we will never arrive at a configuration where d is entailed (unless information is added from the outside). Thus the traces of A all have a functionality which is weaker or equal to that of f. The limit of any trace of A must be a constraint which is a fixpoint of f, otherwise the execution of A would have added more information to the store. Thus, the abstract semantics of A can be given as follows.

$$\mathcal{A}_\Pi[\![A]\!] = \{t \mid \mathrm{fn}(t) \sqsubseteq f, \lim(t) \in f\}$$

This relationship is stated without proof since the further developments do not rely on it, but it is straight-forward to derive a proof from the correctness proof of the fixpoint semantics given in Chapter 9.

5.3 Compositionality of the abstract semantics

In the following sections, we will show that the abstract semantics is compositional. The constructs that need to be considered are conjunction, the existential quantifier, and the selection operator.

5.3.1 Conjunction

Consider the result semantics of a conjunction of agents. The following lemma relates the result semantics of a conjunction of agents to the abstract semantics of the agents. In the proof we take advantage of the fact that whenever $t \in \mathcal{O}_\Pi[\![A]\!]$, for some agent A, there is a trace t' satisfying $v(t')_0 = \bot$ with the same limit and functionality as t, defined by, e.g., $v(t')_0 = \bot$, $v(t')_{i+1} = v(t)_i$, and $i+1 \in r(t')$ iff $i \in r(t)$ for $i \in \omega$.

Lemma 5.3.1 Suppose $\{A^j\}_{j\in I}$ is a countable family of agents. For a constraint c, we have $c \in \mathcal{R}_\Pi[\![\bigwedge_{j\in I} A^j]\!]\bot$ if and only if there is a family of traces $(t_j)_{j\in I}$ such that $t_j \in \mathcal{A}_\Pi[\![A^j]\!]$ and $\lim(t_j) = c$ for $j \in I$, and $\bigcap_{j\in I} \mathrm{fn}(t_j) = c\uparrow$.

Proof. ($\Rightarrow$) Suppose $c \in \mathcal{R}_\Pi[\![\bigwedge_{j\in I} A^j]\!]\bot$. By Proposition 5.2.5 there is an input-free trace $t \in \mathcal{O}[\![\bigwedge_{j\in I} A^j]\!]$ such that $v(t)_0 = \bot$, $\lim(t) = c$, and

$\mathrm{fn}(t) = c\uparrow$. By Lemma 4.7.5 there is for each $j \in I$ a trace $t_j \in \mathcal{O}_\Pi[\![A^j]\!]$ such that $v(t_j) = v(t)$ and $\bigcup_{j\in I} r(t_j) = r(t)$. Using Proposition 5.2.2 it follows that a fixpoint of $\mathrm{fn}(t)$ is also a fixpoint of all $\mathrm{fn}(t_j)$, so $\bigcap_{j\in I} \mathrm{fn}(t_j) = c\uparrow$.

($\Leftarrow$) Each trace t_j connects A^j to a computation $(A^j_i : c^j_i)_{i\in\omega}$. We can assume that $c^j_0 = \bot$, for $j \in I$. We also assume that c is not finite. Let p be a function $p : \omega \to I$ such that for each $j \in I$ there are infinitely many $k \in \omega$ such that $p(k) = j$. We will form a computation $(B_i : d_i)_{i\in\omega}$ of the agent $\bigwedge_{j\in I} A^j$, where each B_i is of the form $\bigwedge_{j\in I} B^j_i$.

Let $B_0 = \bigwedge_{j\in I} A^j_0$. Let $d_0 = \bigsqcup_{j\in I} c^j_0 \quad (= \bot)$.

Suppose $B_k : d_k$ is defined for $k \le n$. We define $B_{n+1} : d_{n+1}$ as follows. Let $k = p(n)$. Let m be the maximal integer such that $A^k_m = B^k_n$ and $c^k_m \sqsubseteq d_n$.

1. If there is a computation step $A^k_m : c^k_m \longrightarrow A^k_{m+1} : c^k_{m+1}$, we make the constructed computation perform a corresponding computation step, by letting $B^j_{n+1} = B^j_n$, for $j \neq k$, $B^k_{n+1} = A^k_{m+1}$, and $d_{n+1} = d_n \sqcup c^k_{m+1}$.
2. If there is no computation step $A^k_m : c^k_m \longrightarrow A^k_{m+1} : c^k_{m+1}$, let $B_{n+1} = B_n$ and $d_{n+1} = d_n$. Note that in this case d_n must be a fixpoint of $\mathrm{fn}(t_k)$ (since c^k_m is a fixpoint of $\mathrm{fn}(t_k)$, and by the way m was selected we know that $c^k_{m+1} \not\sqsubseteq d_n$).

Consider the limit $d = \bigsqcup_{n\in\omega} d_n$. Note that $d_n \sqsubseteq c$ for all n, so $d \sqsubseteq c$. Suppose $d \sqsubset c$. We can see that in the construction of $(B_n : d_n)_{n\in\omega}$, case 1 was only applied a finite number of times for each $j \in I$. This implies that for each $j \in I$, there is an infinite chain

$$d^j_0, \quad d^j_1, \quad d^j_2, \quad \ldots$$

of fixpoints of $\mathrm{fn}(t_j)$. Since the limit of each of these chains is d, and by continuity, d must also be a fixpoint of each $\mathrm{fn}(t_j)$. But then d is a fixpoint of $\bigcap_{j\in I} \mathrm{fn}(t_j)$ and we arrive at a contradiction. $\square$

Using Lemma 5.3.1 it is fairly straight-forward to show that the abstract semantics of a conjunction can be obtained from the agents. In the proof of the theorem below, we will use the fact that for an arbitrary trace, there is an agent whose operational semantics contains the trace. The construction of the agent is as follows.

Definition 5.3.2 For a trace t, let $[t]$ be the agent

$$\bigwedge_{i\in r(t)} (v(t)_i \Rightarrow v(t)_{i+1}) \quad .$$

$\square$

Clearly, $t \in \mathcal{A}_\Pi[\![[t]]\!]$. Moreover, we have the following:

Lemma 5.3.3 Let t be a trace. If $u \in \mathcal{O}_\Pi[\![[t]]\!]$ is a trace of $[t]$, then $\mathrm{fn}(u) \sqsubseteq \mathrm{fn}(t)$.

The lemma follows from the computation rules.

Lemma 5.3.4 Let t be a trace such that $v(t)_0 = \bot$, let $\bar{t}$ be the inverse of t, and let $c = \lim(t)$. Then $\mathrm{fn}(t) \cap \mathrm{fn}(\bar{t}) = c \uparrow$ and $\mathrm{fn}(t) \cup \mathrm{fn}(\bar{t}) = \mathcal{U}$.

The lemma follows immediately from proposition 5.2.2.

Theorem 5.3.5 Let $\{A^j \mid j \in I\}$ be a family of agents. For any trace t,

$$t \in \mathcal{A}_\Pi[\![\bigwedge_{j\in I} A^j]\!]$$

iff for each $j \in I$ there is a $t_j \in \mathcal{A}_\Pi[\![A^j]\!]$, such that $\lim(t) = \lim(t_j)$ and $\mathrm{fn}(t) \supseteq \bigcap_{j\in I} \mathrm{fn}(t_j)$.

Proof. ($\Rightarrow$) Let $t \in \mathcal{A}_\Pi[\![\bigwedge_{j\in I} A^j]\!]$. By definition there is a $t' \in \mathcal{O}_\Pi[\![\bigwedge_{j\in I} A^j]\!]$ such that t is a subtrace of t'. Let $c = \lim(t)$, and let B be the agent

$$B = (\bigwedge_{j\in I} A^j) \wedge [\bar{t}],$$

where $\bar{t}$ is the inverse of the trace t. Since $\mathrm{fn}(t) \cap \mathrm{fn}(\bar{t}) = c \uparrow$, and $\mathrm{fn}(t') \subseteq \mathrm{fn}(t)$, and $c = \lim(t')$, we have by Lemma 5.3.1 that $c \in \mathcal{R}_\Pi[\![B]\!]\bot$. Again, by using the decomposition of B into $[\bar{t}]$ and the individual A^j in Lemma 5.3.1 it follows that there is a trace u of $[\bar{t}]$ and that for each $j \in I$ there is a $t_j \in \mathcal{O}_\Pi[\![A^j]\!]$, such that $\lim(t) = \lim(t_j)$ and $\bigcap_{j\in I} \mathrm{fn}(t_j) \cap \mathrm{fn}(u) = c \uparrow$. By $\mathrm{fn}(\bar{t}) \subseteq \mathrm{fn}(u)$ and $\mathrm{fn}(t) \cup \mathrm{fn}(\bar{t}) = \mathcal{U}$ and $\mathrm{fn}(t) \cap \mathrm{fn}(\bar{t}) = c \uparrow$ we get $\mathrm{fn}(t) \supseteq \bigcap_{j\in I} \mathrm{fn}(t_j)$.

($\Leftarrow$) Suppose that for each $i \in I$ there is a $t_j \in \mathcal{A}_\Pi[\![A^j]\!]$ such that $\lim(t) = \lim(t_j)$ and $\mathrm{fn}(t) \supseteq \bigcap_{j\in I} \mathrm{fn}(t_j)$. By definition there is for each $j \in I$ a $t'_j \in \mathcal{O}_\Pi[\![A^j]\!]$ so that t_j is a subtrace of t'_j. Let B be the agent

$$B = (\bigwedge_{j\in I} A^j) \wedge [\bar{t}].$$

By $\mathrm{fn}(t) \supseteq \bigcap_{j\in I} \mathrm{fn}(t_j)$ and $\mathrm{fn}(t'_j) \subseteq \mathrm{fn}(t_j)$ and the fact that all involved traces have limit c, we infer that $\mathrm{fn}(t) \cap \bigcap_{j\in I} \mathrm{fn}(t'_j) = c \uparrow$. By Lemma 5.3.1 it follows that $c \in \mathcal{R}_\Pi[\![B]\!]\bot$. Again, by using the decomposition of B into $[\bar{t}]$ and the conjunction $\bigwedge_{j\in I} A^j$ in Lemma 5.3.1 it follows that there is a trace t' of $\bigwedge_{j\in I} A^j$ and a trace u of $[\bar{t}]$ such that $\mathrm{fn}(t') \cap \mathrm{fn}(u) = c \uparrow$. Since $\mathrm{fn}(\bar{t}) \subseteq \mathrm{fn}(u)$ and $\mathrm{fn}(t) \cup \mathrm{fn}(\bar{t}) = \mathcal{U}$, we infer that $\mathrm{fn}(t') \subseteq \mathrm{fn}(t)$, i.e., that t is a subtrace of t' which implies that $t \in \mathcal{A}_\Pi[\![\bigwedge_{j\in I} A^j]\!]$. $\square$

5.3.2 The existential quantifier

The treatment of the existential quantifier is certainly the most difficult part of this article. An early version of the thesis gave an incorrect characterisation of the compositionality of the abstract semantics with respect to the existential quantifier. A similar error was made by Saraswat, Rinard and Panangaden [71]. When we look at the semantics of an existentially quantified agent $\exists_X A$, it should be clear that for any trace t of the agent $\exists_X A$ there is a corresponding trace u of the agent A. How are these two traces related? Obviously, since the variable X is hidden, the traces u and t need not agree on the behaviour with respect to X, but for other variables there should be a correspondence between the two traces. It follows that the limit and functionality of t and u should agree when we do not look at how the variable X is treated. Are these requirements sufficient? Well, almost. It turns out that it is necessary to add a third requirement to the trace u. (This requirement is the one that was missing from a previous version of the thesis, and from [71].) For example, consider the agent

$$A = (X = 10 \Rightarrow Y = 7 \,[\!]\, \mathbf{true} \Rightarrow Z = 5).$$

The agent is non-deterministic, since if $X = 10$ it might either produce $Y = 7$, or $Z = 5$. However, the agent

$$\exists_X A$$

is *deterministic* and will always produce the result $Z = 5$. In other words, when we consider the traces of A, we should not consider the traces that contain input steps where X becomes bound. We conclude that the semantics for an agent $\exists_X A$ should only consider the traces of A which do not receive any input on X.

Given that an agent A has a trace u, and that there is a corresponding trace t of $\exists_X A$, how are the functionalities of the traces related? Intuitively, the difference lies in that the functionality of t cannot depend on X, i.e., it cannot detect if X is bound or bind X to a value. These considerations lead to the following definition.

Definition 5.3.6 For a closure operator f, let $\mathrm{E}_X(f)$ be the closure operator defined as follows.

$$\mathrm{E}_X(f) = (\exists_X \circ f \circ \exists_X) \sqcup \mathbf{id}$$

$\square$

Proposition 5.3.7 For a closure operator f, the closure operator $\mathrm{E}_X(f)$ has the set of fixpoints given by the following equation.

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

Proof. Note that for any constraint c, we have $\mathrm{E}_X(f)c \sqsubseteq f(c)$. From this follows that any fixpoint of f must also be a fixpoint of $\mathrm{E}_X(f)$. Let $g = \mathrm{E}_X(f)$.

($\supseteq$) 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 $g(c) = \exists_X(f(\exists_X(c))) \sqcup c = \exists_X(f(\exists_X(d))) \sqcup c \sqsubseteq \exists_X(f(d)) \sqcup c = \exists_X(d) \sqcup c = \exists_X(c) \sqcup c = c$.

($\subseteq$) Now, suppose that c is a fixpoint of g. Let $d = f(\exists_X(c))$. The constraint d is of course a fixpoint of f and since $c = g(c)$, we must have $c \sqsupseteq \exists_X(f(\exists_X(c))) = \exists_X(d)$. So $\exists_X(c) \sqsupseteq \exists_X(d)$, and since $f(\exists_X(c)) \sqsupseteq \exists_X(c)$, which implies $\exists_X(c) \sqsubseteq \exists_X(d)$, we have $\exists_X(c) = \exists_X(d)$. □

In the proof of the compositionality theorem, the following proposition will be useful. Note that for a trace t, the traces of $\mathcal{A}_\Pi[\![\exists_X[t]]\!]$ are the traces which have a functionality weaker than the one of t and a limit which is a fixpoint of the functionality of t.

Proposition 5.3.8 Given an agent A, let u be a trace in $\mathcal{A}_\Pi[\![A]\!]$ such that $(\mathrm{fn}\, u)(\exists_X(\lim u)) = \lim u$. Let t be a trace such that $\mathrm{fn}\, t \sqsubseteq \mathrm{E}_X(\mathrm{fn}\, u)$ and $\exists_X(\lim t) = \exists_X(\lim u)$. It follows that $\lim u \in \mathcal{R}_\Pi[\![A \wedge \exists_X[\bar{t}]]\!]$.

Proof. Note that $u \in \mathcal{A}_\Pi[\![A]\!]$ and there is a trace $t_0 \in \mathcal{A}_\Pi[\![\exists_X[\bar{t}]]\!]$ such that $\mathrm{fn}\, t_0 = \mathrm{E}_X(\mathrm{fn}\, \bar{t})$ and $\lim u = \lim t_0$. By Lemma 5.3.1 it is sufficient to show that $(\mathrm{fn}\, u) \cap (\mathrm{E}_X(\mathrm{fn}\, \bar{t})) = (\bot \to \lim u)$. We compute the least fixed point of $(\mathrm{fn}\, u) \cap (\mathrm{E}_X(\mathrm{fn}\, \bar{t}))$ by forming the chains $d_0, d_1, d_2, \ldots$ and $e_0, e_1, e_2, \ldots$ as follows.

1. $d_0 = e_0 = \bot$.
2. For i even, let
   (a) $d_{i+1} = \mathrm{E}_X(\mathrm{fn}\, u)d_i$, and
   (b) $e_{i+1} = (\mathrm{fn}\, u)e_i$.
3. For i odd, let
   (a) $d_{i+1} = (\mathrm{fn}\, \bar{t})d_i$ and
   (b) $e_{i+1} = \mathrm{E}_X(\mathrm{fn}\, \bar{t})e_i$.

It is straight-forward to show that $\exists_X d_i = \exists_X e_i$, for all $i \in \omega$. Let $d = \sqcup_{i\in\omega} d_i$ and $e = \sqcup_{i\in\omega} e_i$. We want to show that $d = \lim t$. Suppose $d \sqsubset \lim t$. By continuity, $d \in \mathrm{fn}\, \bar{t}$ and $d \in \mathrm{E}_X(\mathrm{fn}\, u)$. Thus $d \notin \mathrm{fn}\, t$ which implies that $d \notin \mathrm{E}_X(\mathrm{fn}\, u)$. We have arrived at a contradiction.

It follows that $d = \lim t$. By assumption we have $\exists_X \lim t = \exists_X \lim u$ and $(\mathrm{fn}\, u)(\exists_X \lim u) = \lim u$. It follows immediately that $(\mathrm{fn}\, u)(\exists_X d) = \lim u$. We can conclude that $e = (\mathrm{fn}\, u)(\exists_X d) = \lim u$. □

Theorem 5.3.9 For an agent A and a variable X there is a trace $t \in \mathcal{A}_\Pi[\![\exists_X A]\!]$ iff there is a $u \in \mathcal{A}_\Pi[\![A]\!]$ such that $\lim(u) = (\mathrm{fn}(u) \circ \exists_X)\lim(u)$, $\exists_X(\lim(t)) = \exists_X(\lim(u))$ and $\mathrm{fn}(t) \sqsubseteq \mathrm{E}_X(\mathrm{fn}(u))$.

Proof. ($\Rightarrow$) Suppose $t \in \mathcal{O}_\Pi[\![\exists_X A]\!]$. By Proposition 4.7.6, there must be a trace $u \in \mathcal{O}_\Pi[\![A]\!]$, such that with $v(t) = (d_i)_{i\in\omega}$ and $v(u) = (e_i)_{i\in\omega}$ it holds that $r(t) = r(u)$; $e_0 = \exists_X(d_0)$; $d_{i+1} = d_i \sqcup \exists_X(e_{i+1})$, for $i \in r(t)$; and $e_{i+1} = e_i \sqcup \exists_X(d_{i+1})$, for $i \in \omega \setminus r(t)$. It is straight-forward to prove by induction that for all $i \in \omega$, $\exists_X(d_i) = \exists_X(e_i)$, and thus, $\exists_X(\lim(t)) = \exists_X(\lim(u))$.

Next we show that $\mathrm{fn}(t) \sqsubseteq \mathrm{E}_X(\mathrm{fn}(u))$. Let i be fixed such that $i \in r(t)$. It is sufficient to show that $(d_i \to d_{i+1}) \sqsubseteq \mathrm{E}_X(\mathrm{fn}(u))$. Note that for all $i \in \omega$, $e_i \sqsubseteq \mathrm{fn}(u)(\exists_X(d_i))$ (this is easily proved by induction). If c is a constraint such that $c \sqsupseteq d_i$, we have $e_i \sqsubseteq \mathrm{fn}(u)(\exists_X(d_i)) \sqsubseteq \mathrm{fn}(u)(\exists_X(c))$, and thus $\mathrm{fn}(u)(\exists_X(c)) \sqsupseteq e_{i+1}$, from which follows that $\mathrm{E}_X(\mathrm{fn}(u))c \sqsupseteq d_{i+1}$, since by the reduction rules $d_{i+1} = d_i \sqcup \exists_X(e_{i+1})$.

To show that $\lim(u) = (\mathrm{fn}(u) \circ \exists_X)\lim(u)$, we first note that $\mathrm{fn}(u)e_i \sqsubseteq \lim(u)$, for all $i \in \omega$, from which follows that $(\mathrm{fn}(u) \circ \exists_X)e_i \sqsubseteq \lim(u)$, for all i, and thus $(\mathrm{fn}(u) \circ \exists_X)\lim(u) \sqsubseteq \lim(u)$. By the argument in the previous paragraph we have $e_i \sqsubseteq \mathrm{fn}(u)(\exists_X(d_i))$, for $i \in \omega$, and since $\exists_X(d_i) = \exists_X(e_i)$ we also have $e_i \sqsubseteq \mathrm{fn}(u)(\exists_X(e_i))$, for all i. By continuity we have $\lim(u) \sqsubseteq \mathrm{fn}(u)(\exists_X(\lim(u)))$.

($\Leftarrow$) Suppose that $u \in \mathcal{O}_\Pi[\![A]\!]$ such that $\lim(u) = (\mathrm{fn}(u) \circ \exists_X)\lim(u)$. Suppose also that the trace t is such that $\lim t = \lim u$ and $\mathrm{fn}\, t \sqsubseteq \mathrm{E}_X(\mathrm{fn}\, u)$. We want to show that $t \in \mathcal{A}_\Pi[\![\exists_X A]\!]$.

By the Proposition 5.3.8 there is a fair, input-free computation $(A_i \wedge \exists_X^{c_i} B_i : e_i)_{i\in\omega}$ where the configuration $A_0 \wedge \exists_X^{c_0} B_0 : e_0$ is equal to $A \wedge \exists_X[\bar{t}] : \bot$ and $\sqcup_{i\in\omega} e_i = \lim u$. Let u' be the trace corresponding to the computation $(A_i : e_i)_{i\in\omega}$. We have, by the computation rules,

1. $A_i : e_i \longrightarrow A_{i+1} : e_{i+1}$ and $B_i : c_i = B_{i+1} : c_{i+1}$, if $i \in r(u')$, and

2. $B_i : c_i \sqcup (\exists_X e_i) \longrightarrow B_{i+1} : c_{i+1}$, $A_{i+1} = A_i$ and $e_{i+1} = e_i \sqcup (\exists_X c_{i+1})$ if $i \notin r(u')$.

Let the chain $d_0, d_1, \ldots$ be as follows.

1. $d_0 = \bot$.

2. $d_{i+1} = c_{i+1} \sqcup \exists_X e_{i+1}$, if $i \in r(u)$.

3. $d_{i+1} = c_{i+1}$, if $i \notin r(u)$.

It is straight-forward to establish that for $i \in r(u)$, $d_{i+1} = d_i \sqcup \exists_X(e_{i+1})$, and for $i \notin r(u)$, $e_{i+1} = e_i \sqcup \exists_X(d_{i+1})$. We can now form a trace t' with $r(t') = r(u)$ and $v(t') = (d_i)_{i\in\omega}$ such that, by Proposition 4.7.6, $t' \in \mathcal{O}_\Pi[\![\exists_X A]\!]$.

We also would like to show that t is a subtrace of t'. Consider the computation $(B_i : c_i \sqcup \exists_X e_i)_{i\in\omega}$ of $[t]$. Clearly this is the same as $(B_i : d_i \sqcup \exists_X e_i)_{i\in\omega}$. Note that the trace of this computation is $\bar{t'}$. It follows that $\mathrm{fn}\, t' \sqsupseteq \mathrm{fn}\, t$ and that $\lim t' = \lim t$.

□

5.3.3 The selection operator.

Theorem 5.3.10 For $n \geq 0$, agents $A_1, \ldots, A_n$ and constraints $c_1, \ldots, c_n$, we have a trace $t \in \mathcal{A}_\Pi[\![(c_1 \Rightarrow A_1 [] \ldots [] c_n \Rightarrow A_n)]\!]$ if and only if either

1. there is a $k \leq n$ and $u \in A_\Pi[\![A_k]\!]$ such that $\lim(t) = \lim(u) \sqsupseteq c_k$ and $\mathrm{fn}(t) \sqsubseteq (c_k \rightarrow \mathrm{fn}(u))$, or
2. $\mathrm{fn}(t) = \mathbf{id}$ and $c_k \not\sqsubseteq \lim(t)$, for $k \leq n$.

Proof. ($\Rightarrow$) Suppose $t \in A_\Pi[\![(c_1 \Rightarrow A_1 [] \ldots [] c_n \Rightarrow A_n)]\!]$. There is a trace $t' \in \mathcal{O}_\Pi[\![(c_1 \Rightarrow A_1 [] \ldots [] c_n \Rightarrow A_n)]\!]$ such that t is a subtrace of t'.

Suppose that $\lim(t) \sqsupseteq c_l$, for some $l \leq n$. By Proposition 4.7.7 there is, for some $k \leq n$, a trace $u \in \mathcal{O}_\Pi[\![A_k]\!]$ such that $\lim(u') = \lim(t')$ and $\mathrm{fn}(t') = (c_k \rightarrow \mathrm{fn}(u))$. The trace u is of course also a trace of $\mathcal{A}_\Pi[\![(c_1 \Rightarrow A_1 [] \ldots [] c_n \Rightarrow A_n)]\!]$. Since $\mathrm{fn}(t) \sqsubseteq \mathrm{fn}(t)$, we have $\mathrm{fn}(t) \sqsubseteq (c_k \rightarrow \mathrm{fn}(u))$.

Suppose that there are no $l \leq n$ such that $\lim(t) \sqsupseteq c_l$. By Proposition 4.7.7 we have $r(t) = \emptyset$, and thus $\mathrm{fn}(t) = \mathbf{id}$.

($\Leftarrow$) Suppose $u \in \mathcal{A}_\Pi[\![A_k]\!]$, and that t is a trace such that $\lim(t) = \lim(u) \sqsupseteq c_k$, and $\mathrm{fn}(t) \sqsubseteq (c_k \rightarrow \mathrm{fn}(u))$. We have immediately that u is a subtrace of a trace $u' \in \mathcal{O}_\Pi A_k$. By Proposition 4.7.7 there is a trace $t' \in \mathcal{O}[\![(c_1 \Rightarrow A_1 [] \ldots [] c_n \Rightarrow A_n)]\!]$ such that $\lim(t') = \lim(u')$ and $\mathrm{fn}(t') = (c_k \rightarrow \mathrm{fn}(u'))$. Thus, $\mathrm{fn}(t') \sqsupseteq (c_k \rightarrow \mathrm{fn}(u)) \sqsupseteq \mathrm{fn}(t)$, and we conclude t is a subtrace of t'.

Let t be a trace such that there are no $l \leq n$ such that $\lim(t) \sqsupseteq c_l$. By Proposition 4.7.7 it follows that $t \in \mathcal{O}_\Pi(c_1 \Rightarrow A_1 [] \ldots [] c_n \Rightarrow A_n)$, and thus t also belongs to the abstract semantics of the selction. □

5.4 The abstract semantics is fully abstract

A semantics is *fully abstract* if the semantics does not give more information than what is necessary to distinguish between agents that behave differently when put into a context.

Theorem 5.4.1 Suppose we have agents A and A'. If $\mathcal{A}_\Pi[\![A]\!] \neq \mathcal{A}_\Pi[\![A']\!]$ then there is an agent B such that $\mathcal{R}_\Pi[\![A \wedge B]\!] \neq \mathcal{R}_\Pi[\![A' \wedge B]\!]$.

$$
\boxed{
\begin{aligned}
\mathcal{A}_\Pi\llbracket c\rrbracket &= \{t \,|\, \mathrm{fn}(t) \supseteq (\bot \to c) \text{ and } \lim(t) \sqsupseteq c\} \\
\mathcal{A}_\Pi\llbracket \textstyle\bigwedge_{j\in I} A^j\rrbracket &= \{t \,|\, t_j \in \mathcal{A}_\Pi\llbracket A^j\rrbracket \text{ and } \lim(t_j) = \lim(t), \text{ for } j \in I, \\
&\qquad \mathrm{fn}(t) \supseteq \textstyle\bigcap_{j\in I} \mathrm{fn}(t_j)\} \\
\mathcal{A}_\Pi\llbracket \exists_X A\rrbracket &= \{t \,|\, t' \in \mathcal{A}_\Pi\llbracket A\rrbracket, \\
&\qquad \exists_X(\lim(t)) = \exists_X(\lim(t')), \\
&\qquad \lim(t') = (\mathrm{fn}(t') \circ \exists_X)\lim(t'), \text{ and} \\
&\qquad \mathrm{fn}(t) \supseteq \mathrm{E}_X(\mathrm{fn}(t'))\} \\
\mathcal{A}_\Pi\llbracket \textstyle[]_{k\le n}\, c_k \Rightarrow A_k\rrbracket &= \{t \,|\, \mathrm{fn}(t) = \mathbf{id} \text{ and } \lim(t) \not\sqsupseteq c_k, \text{ for } k \le n\} \textstyle\bigcup \\
&\quad\ \{t \,|\, k \le n, t' \in \mathcal{A}_\Pi\llbracket A_k\rrbracket, \\
&\qquad \lim(t) = \lim(t') \sqsupseteq c_k, \text{ and} \\
&\qquad \mathrm{fn}(t) = (c_k \to \mathrm{fn}(t'))\} \\
\mathcal{A}_\Pi\llbracket p(X)\rrbracket &= \ \mathcal{A}_\Pi\llbracket A[X/Y]\rrbracket, \\
&\qquad \text{where the definition of } p \text{ is } p(Y) :: A
\end{aligned}
}
$$

Figure 5.1: The abstract semantics in equational form.

Proof. Suppose $t \in \mathcal{A}_\Pi\llbracket A\rrbracket \setminus \mathcal{A}_\Pi\llbracket A'\rrbracket$. Consider the agent $A \wedge [\bar{t}]$. By Lemma 5.3.1 we have $\lim(t) \in \mathcal{R}_\Pi\llbracket A \wedge [\bar{t}]\rrbracket\bot$. Suppose $\lim(t) \in \mathcal{R}_\Pi\llbracket A' \wedge [\bar{t}]\rrbracket\bot$. By Lemma 5.3.1 there are traces $t_1 \in \mathcal{O}_\Pi\llbracket A'\rrbracket$ and $t_2 \in \mathcal{O}_\Pi\llbracket[\bar{t}]\rrbracket$ such that $\lim(t_1) = \lim(t_2) = \lim(t)$ and $\mathrm{fn}(t_1) \cap \mathrm{fn}(t_2) = c\uparrow$. Since clearly $\mathrm{fn}(t_2) \supseteq \mathrm{fn}(\bar{t})$, this implies that $\mathrm{fn}(t_1) \subseteq \mathrm{fn}(t)$, so t must be a subtrace of t_1. This contradicts the assumption that $t \notin \mathcal{O}_\Pi\llbracket A'\rrbracket$. □

5.5 The Abstract Semantics in Equational Form

As we have established that that the abstract semantics is compositional, we can give the abstract semantics as a set of equations (Figure 5.1).

5.6 A proof of full abstraction using finite programs

Our proof of full abstraction in Theorem 5.4.1 relied on the use of infinite conjunctions to express an agent that could produce an 'infinite' trace. Is it possible to give a proof of full abstraction that does not use infinite conjunctions? It turns out that if we make some very reasonable assumptions about the constraint system, and extend the result semantics to cope with infinite input, it is possible to give a proof of full abstraction that does not use infinite conjunctions.

The proof in this section resembles a proof of full abstraction given by Russell [68]. The idea is that we assume that a representation of a trace is provided as input. It is then possible to write a procedure that 'interprets' the trace and thus exhibits a behaviour similar to the agent $[\bar{t}]$ in the previous proof. We must make some assumptions about the constraint system. First, we assume that it is sufficiently powerful to emulate itself. That is, we assume that there is in the domain of values a representation of each finite constraint. We also assume that it is possible to write procedures that can take a representation of a finite constraint and can interpret its behaviour. Third, we assume that the term model is a part of the constraint system and that there are some appropriate function symbols.

5.6.1 The generalised result semantics

We previously defined the result semantics of an agent, $\mathcal{R}_\Pi[\![A]\!]c$ only for finite inputs c. This restriction was introduced to make it possible to give the input in the first configuration of a computation, as any intermediate state of a computation must be finite. However, if we allow the input to be given during the course of a computation, we can consider a generalised version of the result semantics that also allows infinite inputs.

For an agent A, a program Π, and a constraint c the *generalised result semantics*

$$\begin{aligned}\mathcal{R}_\Pi[\![A]\!]c = \{\lim(t) \mid\ & t \in \mathcal{O}_\Pi[\![A]\!],\\ & v(t)_0 \sqsubseteq c,\\ & c_{i+1} \sqsubseteq c_i \sqcup c,\ \text{for } i \in \omega \setminus r(t),\ \text{and}\\ & \lim(t) \sqsupseteq c\}\end{aligned}$$

It is easy to see that for finite constraints the generalised result semantics conforms with the first result semantics, and that for infinite constraints the generalised result semantics gives the result produced by an agent when it receives infinite input.

Proposition 5.6.1 For an agent A, and constraints c and d, we have $c \in \mathcal{R}_\Pi[\![A]\!]d$ iff there is a trace $t \in \mathcal{A}_\Pi[\![A]\!]$ such that $\lim(t) = c$, and $\mathrm{fn}(t) \cap (\bot \to d) = (\bot \to c)$.

5.6.2 Can a ccp language interpret its constraint system?

The answer, for any reasonable constraint system, is yes. But first we must define what it means for a constraint system to be self-interpretable.

First, there must be a way to represent the finite constraint as values in the constraint system. For example, it the term model we can of course represent the finite constraints as terms.

Second, for each finite constraint we need a way to bind a variable to that finite constraint, i.e., a finite constraint that does precisely that. In the term model this is easily accomplished, since we have constraints that can bind a variable to any term.

Of course, we also need to be able to 'interpret' the representations of constraints as real constraints, i.e., as tell and ask constraints. So we also require that it should be possible to write procedures that interprets representations of constraints and emulates the behaviour of the corresponding ask and tell constraints (these are requirements three and four). In the term model, implementing these procedures is a straight-forward programming task.

Definition 5.6.2 A constraint system is *self-interpretable* if the following holds.

1. There is an injective map l from finite constraints to the domain of values.

2. For each finite constraint c and variable X there is a constraint $X = l(c)$ which binds X to $l(c)$.

3. For any fixed set of variables $X_1, \ldots, X_n$ it is possible to define a procedure entail, such that a call entail$(R, F, X_1, \ldots, X_n)$ will, when R is bound to $l(c)$ and c is a constraint that depends only on the variables $X_1, \ldots, X_n$, and c is entailed, bind F to 1.

4. For $X_1, \ldots, X_n$ as above it is possible to define a procedure toStore such that a call toStore$(R, X_1, \ldots, X_n)$ will, when $R = l(c)$ and c is a constraint that depends only on the variables $X_1, \ldots, X_n$, add the constraint c to the store.

□

One would normally expect a constraint system to be implementable on a computer, and to be sufficiently powerful to implement the set of computable functions. Given this it is not a big step to assume a constraint system to be self-interpretable. As representations of the finite constraints, it is of course very natural to consider the elements of the term model.

5.6.3 Giving the representation of a trace

We need a way to construct a constraint that gives a representation of a trace. Suppose that t is a trace which only depends on variables $X_1, \ldots, X_n$. We must construct a constraint c, we will later refer to it as $[t]$, which binds a variable L to a list representation of the trace t. Let $(d_i)_{i\in\omega} = v(t)$. Let E_0 be $\mathrm{in}(Z_0)$. For $i-1 \in r(t)$, let E_i be the expression $\mathrm{out}(Z_i)$, and for $i-1 \in \omega \setminus r(t)$, let E_i be $\mathrm{in}(Z_i)$. For $i \in \omega$, let

$$\begin{aligned} c_i \quad \text{be} \quad \exists_{L'}\exists_{Z_0}\ldots\exists_{Z_i}(&L = [E_0, \ldots, E_i \mid L'] \\ &\wedge Z_0 = l(d_0) \wedge \ldots \wedge Z_i = l(d_i)). \end{aligned}$$

Clearly, all c_is are finite constraints, and with $c(t) = \sqcup_{i\in\omega} c_i$, $c(t)$ is the constraint which binds L to a representation of the trace t.

5.6.4 Interpreting traces

Next we construct a procedure $\mathrm{interpret}(L, X_1, \ldots, X_n)$, that given a list representation of a trace t that only depends on the variables $X_1, \ldots, X_n$, behaves like the agent $[t]$ in the previous proof of full abstraction. We want the call $\mathrm{interpret}(L, X_1, \ldots, X_n)$ to be such that for traces

$$u \in \mathcal{A}_\Pi[\![\mathrm{interpret}(L, X_1, \ldots, X_n)]\!]$$

we have $\mathrm{fn}(u) \sqsubseteq \mathrm{fn}(t)$ whenever $\exists_L(\lim(u)) = \lim(t)$ and L is bound to the list representation in the constraint $\lim(u)$.

We first give interpret in the form of a clp program, since this version may be easier to read than the ccp version.

```
$\mathrm{interpret}([\mathrm{out}(R) \mid L], X_1, \ldots, X_n) :-$
        $\mathrm{toStore}(R, X_1, \ldots, X_n),$
        $\mathrm{interpret}(L, X_1, \ldots, X_n).$
$\mathrm{interpret}([\mathrm{in}(R) \mid L], X_1, \ldots, X_n) :-$
        $\mathrm{entail}(R, F, X_1, \ldots, X_n),$
        $\mathrm{interpret\_aux}(F, L, X_1, \ldots, X_n).$
$\mathrm{interpret\_aux}(1, L, X_1, \ldots, X_n) :- \mathrm{interpret}(L, X_1, \ldots, X_n).$
```

The same program in ccp.

```
$\mathrm{interpret}(L, X_1, \ldots, X_n) ::$
    $((\exists_R\exists_{L'} L = [\mathrm{out}(R) \mid L'])$
        $\Rightarrow \exists_R\exists_{L'}(L = [\mathrm{out}(R) \mid L']$
                $\wedge\, \mathrm{toStore}(R, X_1, \ldots, X_n)$
                $\wedge\, \mathrm{interpret}(L', X_1, \ldots, X_n))$
    $[]\ \exists_R\exists_{L'}(L = [\mathrm{in}(R) \mid L'])$
        $\Rightarrow \exists_R\exists_{L'}\exists_F(L = [\mathrm{in}(R) \mid L']$
                $\wedge\, \mathrm{entail}(R, F, X_1, \ldots, X_n)$
                $\wedge\, (F = 1 \Rightarrow \mathrm{interpret}(L', X_1, \ldots, X_n))))$
```

Does interpret behave as intended? Suppose that L does eventually get bound to the list representation of the trace t. The functionality of any trace of the call interpret$(L, X_1, \ldots, X_n)$ is at most f_0, where f_i, $i \in \omega$, is given as follows (recall that $(d_i)_{i\in\omega} = v(t)$).

$$f_i = \begin{cases} f_{i+1}, & \text{if } i \notin r(t) \\ (d_i \to d_{i+1}) \cap f_{i+1}, & \text{if } i \in r(t) \end{cases}$$

It is easy to establish that $f_0 = \mathrm{fn}(t)$.

5.6.5 The proof

Now we are ready to give the alternative proof of full abstraction. We will assume that the constraint system is self-interpretable, and contains the term model, where the set of function symbols includes the list constructor $[\cdot \mid \cdot]$, in$(\cdot)$ and out$(\cdot)$. Recall that A and A' are assumed to be agents such that $t \in \mathcal{A}_\Pi[\![A]\!]$ but $t \notin \mathcal{A}_\Pi[\![A']\!]$. We want to show that there is an agent B and some constraint c such that $\mathcal{R}'_\Pi[\![A \wedge B]\!]c \neq \mathcal{R}'_\Pi[\![A' \wedge B]\!]c$. We will assume that the agents A and A' do not contain any infinite conjunctions, and that thus the set of variables that A and A' depend on are among $X_1, \ldots, X_n$, and that L is not among these variables.

Let c be the constraint $[\bar{t}]$. Let B be the agent interpret$(L, X_1, \ldots, X_n)$ and Π' the program Π extended with definitions of entail, toStore and interpret.

Clearly, we have $d \in \mathcal{R}_{\Pi'}[\![A \wedge B]\!]c$, where $d = \lim t \sqcup c$.

Suppose $d \in \mathcal{R}_{\Pi'}[\![A' \wedge B]\!]c$. There are corresponding traces $t_1 \in \mathcal{O}_{\Pi'}[\![A']\!]$ and $t_2 \in \mathcal{O}_{\Pi'}[\![B]\!]$ such that $\lim t_1 = \lim t_2 = d$ and $\mathrm{fn}(t_1) \cap \mathrm{fn}(t_2) \cap (\bot \to c) = (\bot \to d)$. Let $(e_i)_{i\in\omega} = v(t_1) = v(t_2)$. If we consider traces t'_1 and t'_2 where $v(t'_1) = v(t'_2) = (\exists_L e_i)_{i\in\omega}$ and $r(t'_1) = r(t_1)$ and $r(t'_2) = r(t_2)$ it follows that $t'_1 \in \mathcal{O}_{\Pi'}[\![A']\!]$ (this is easy to establish from the computation rules) and that $\mathrm{fn}(t'_2) \supseteq \mathrm{fn}(\bar{t})$. Thus, $\mathrm{fn}(t'_1) \subseteq \mathrm{fn}(t)$, so t must be a subtrace of t'_1. This implies that $t \in \mathcal{A}_{\Pi'}[\![A']\!]$, which contradicts the assumption that $t \notin \mathcal{A}_\Pi[\![A']\!]$.

5.7 Algebraic properties of concurrent constraint programming

As we have developed a fully abstract semantics of concurrent constraint programming we have a good opportunity to study the algebraic properties of ccp. It turns out that the algebra of concurrent constraint programming agents satisfies the axioms of intuitionistic linear algebra (see Troelstra [79, chapter 8] and Ono [59]), which suggests a relationship between concurrent constraint programming and intuitionistic linear logic.

The work presented in this section is influenced by the results of Mendler, Panangaden, Scott and Seely [52], who show that a semantic model of concurrent constraint programming forms a hyperdoctrine [45, 73], a category-theoretic structure which represents the proof-theoretic structure of logics. Thus, proving that ccp is a hyperdoctrine implies that ccp in fact forms a logic. The authors of reference [52] stress the point; ccp *is* logic.

Other attempts to relate algebraic rules and concurrency include the work by Bergstra and Klop [7], in which algebraic rules were used to define a concurrent language, and Winskel and Nielsen [83], who relate different models of concurrency by examining their category-theoretic properties. Abramsky and Vickers [3] propose the use of quantales as a framework for the study of various aspects of concurrency. (The algebra of quantales is closely related to intuitionistic linear algebra.)

We begin by giving the axiomatic definition of intuitionistic linear algebra, following Troelstra [79]. Intuitionistic linear algebra is to linear logic as Boolean algebra is to propositional logic, i.e., an algebraic formulation of the derivation rules of the logic.

Definition 5.7.1 An *IL-algebra* (intuitionistic linear algebra) is a structure $(X, \sqcup, \sqcap, \bot, \multimap, *, \mathbf{1})$ such that the following holds.

1. $(X, \sqcup, \sqcap, \bot)$ is a lattice.
2. $(X, *, \mathbf{1})$ is a commutative monoid.
3. If $x \leq x'$ and $y \leq y'$ it follows that $x * y \leq x' * y'$ and $x' \multimap y \leq x \multimap y'$.
4. $x * y \leq z$ iff $x \leq y \multimap z$.

□

In the definition, $*$ is to be seen as the multiplicative conjunction and $\sqcap$ as the additive conjunction.

Next we will see how the semantic domain of the fully abstract semantics can be seen as an IL-algebra. Let A consist of the subtrace-closed sets of traces. The lattice-structure of A is simply the inclusion-ordering of the sets of traces.

Proposition 5.7.2 A forms a complete distributive lattice.

Proof. It is easy to see that for any family of subtrace-closed sets, the union and intersection of these sets is also subtrace-closed. From this follows also that A is a distributive lattice. □

Let $* : \mathsf{A} \times \mathsf{A} \to \mathsf{A}$ be parallel composition, i.e.,

$$x * y = \{t \mid t_1 \in x, t_2 \in y, \lim(t_1) = \lim(t_2), \mathrm{fn}(t) \supseteq \mathrm{fn}(t_1) \cap \mathrm{fn}(t_2)\}.$$

Let $\mathbf{1}$ be the set of passive traces, i.e., $\mathbf{1} = \{t \mid r(t) = \emptyset\}$. It is easy to see that $\mathbf{1}$ corresponds to the agent **true**, i.e, the tell constraint which always holds.

Proposition 5.7.3 $(\mathsf{A}, *, \mathbf{1})$ is a commutative monoid. For x and $\{y_i\}_{i\in I} \in \mathsf{A}$ the distributive law $x * (\bigcup_{i\in I} y_i) = \bigcup_{i\in I} x * y_i$ holds,

Define $\multimap$: $\mathsf{A} \times \mathsf{A} \to \mathsf{A}$ according to $x \multimap y = \bigcup\{z \mid x * z \subseteq y\}$. It follows that $x \multimap \cdot$ is an upper adjoint of $\cdot * x$, i.e., that $x \subseteq y \multimap z$ if and only if $x * y \subseteq z$ holds for $x, y, z \in \mathsf{A}$.

We also define an upper adjoint to $\cap$, even though this is not necessary to satisfy the axioms of IL-algebras. Let $\gg$: $\mathsf{A} \times \mathsf{A} \to \mathsf{A}$ according to $x \gg y = \bigcup\{z \mid x \cap z \subseteq y\}$ gives us $x \subseteq y \gg z$ if and only if $x \cap y \subseteq z$, for $x, y, z \in \mathsf{A}$.

It follows immediately that the structure we have obtained satisfies the axioms of IL-algebras.

Theorem 5.7.4 $(\mathsf{A}, \cup, \cap, \emptyset, \multimap, *, \mathbf{1})$ as defined above is an IL-algebra.

Next, we will take a look at how selection in ccp can be expressed using the operations of IL-algebra.

First, note that the functions $\multimap$ and $\gg$ can be expressed directly in terms of sets of traces. For traces t_1, t_2, let $t_1 \vee t_2$ be defined when $v(t_1) = v(t_2)$, and $u = t_1 \vee t_2$ be such that $v(u) = v(t_1)$, and $r(u) = r(t_1) \cup r(t_2)$. We find that

$$x \multimap y = \{t \mid \text{if } u \in x \text{ and } t \vee u \text{ is defined, we have } t \vee u \in y\}.$$

If there were no restriction that the elements of A must be subtrace-closed, $x \gg y$ would consist of the traces which do not belong to x, together with the traces of y, similar to the usual definition of implication in classical logic. But since the complement of an element of A in general is not subtrace-closed, the traces of $x \gg y$ are instead given by

$$x \gg y = \{t \mid \text{if } u \in x \text{ is a subtrace of } t, \text{ then } u \in y\}.$$

For $x \in \mathsf{A}$, let the negation $\sim\! x$ be given as $\sim\! x = x \multimap \emptyset$. The negation of x can also be given directly as a set of traces according to

$$\sim\! x = \{t \mid \text{ there is no } u \in x \text{ such that } \lim(t) = \lim(u)\}.$$

For an agent which is a tell constraint c, the set of traces is

$$c = \{t \mid \lim(t) \sqsupseteq c, \mathrm{fn}(t) \sqsubseteq (\bot \to c)\},$$

writing c for the set of traces of the tell constraint c. Also, note that

$$c \gg \mathbf{1} = \{t \mid \mathrm{fn}(t) \cup (\bot \to c) = \mathcal{U}\}.$$

For a tell constraint c, $\sim\sim\! c$ is the set of traces with limit at least c.

For $a \in \mathsf{A}$, the expression

$$a \cap (c \Rightarrow \mathbf{1}) \cap {\sim\sim} c$$

gives the set of traces t of a which satisfy $\lim t \sqsupseteq c$ and $\operatorname{fn} t = (c \rightarrow \operatorname{fn}(t))$, i.e., the set of traces corresponding to the alternative $c \Rightarrow A$ in a selection.

Given constraints c_1 and c_2, the expression

$${\sim} c_1 \cap {\sim} c_2 \cap 1$$

corresponds to the set of traces in which neither c_1 nor c_2 ever become entailed by the store.

The set of traces of a selection $(c_1 \Rightarrow A_1 \;[]\; c_2 \Rightarrow A_2)$ can thus be given by the expression

$$\begin{aligned}(a_1 \cap (c_1 \Rightarrow \mathbf{1}) \cap {\sim\sim} c_1) \cup (a_2 \cap (c_2 \Rightarrow \mathbf{1}) \cap {\sim\sim} c_2)& \\ \cup ({\sim} c_1 \cap {\sim} c_2 \cap \mathbf{1}),&\end{aligned}$$

where a_k is the set of traces given by the abstract semantics of A_k, for $k \in \{1, 2\}$. (This translation can easily be generalised to selections with an arbitrary number of alternatives.) So non-deterministic selection can be defined using operations derived from parallel composition and the inclusion-ordering of sets of traces.

Chapter 6

A Fully Abstract Semantics for Non-deterministic Data Flow

In the previous chapter we considered a fully abstract semantics for concurrent constraint programming. Since the data flow computation model can be seen as a special case of concurrent constraint programming it follows immediately that it is possible to use the described techniques to give a semantics for data flow languages. In this chapter we will present a fully abstract semantics for data flow based on the fully abstract semantics for ccp. The semantic model will be presented without proofs of compositionality or of full abstraction; it is hoped that the similarity with the semantics for ccp will be sufficient to convince the reader that the model is correct.

A data flow program is a directed graph where the nodes are computational elements (processes) and the edges are communication channels. In the computational models described by Brinch Hansen [8] and Kahn [37] the nodes are imperative programs with explicit computational actions, but as Kahn points out, the nodes (which in his model are always deterministic) can be seen as continuous functions from input to output. Dennis [25] presented a style of data flow programming with a small, predefined set of nodes. The language defines a small set of nodes, from which the programmer is supposed to construct programs; it is not possible to invent new nodes. Also in this case, the nodes can be seen as continuous functions from input to output. As Kahn showed, the semantics of a deterministic data flow program can be given as a continuous function from input sequences to output sequences.

To see how a deterministic node in a data flow network can be seen as a continuous function, first note that the output history of a deterministic node is uniquely given by the input history. The data transmitted along an edge can be represented as a (finite or infinite) string, and sending more data means adding tokens to the end of the string.

If we order the strings in the prefix ordering, we see that the nodes correspond to monotone functions since a node may produce more output (thus extending the output string) when receiving additional input, but not change any output already produced. If we include the infinite strings, so that the partial order of finite and infinite strings forms a cpo, we see that the nodes actually correspond to continuous functions, since the output from a node, when given an infinite input, can be given as a limit of the outputs resulting from the node receiving finite prefixes of the input. For example, if the input is $a_0, a_1, \ldots$ and the output is $b_0, b_1, \ldots$, we can find, for each prefix $b_0, b_1, \ldots, b_m$ of the output, some $n \geq 0$ such that the input sequence $a_0, a_1, \ldots, a_n$ gives an output sequence that begins with $b_0, b_1, \ldots, b_m$. Thus, we can determine what a (deterministic) node will produce for an infinite input, if we know what it produces for finite input.

6.1 Examples

We use the notation $a.s$ for the string constructed by appending the token a to the left of the string s. Thus, we write $1.2.3.\epsilon$ for the string of 1, 2 and 3. When there is no risk of confusion we will sometimes write a string such as the one above in the briefer form 123.

As an example of a deterministic node, consider a node that reads a stream of integers and outputs a stream of integers containing the sum of the two most recently read integers. In the notation used by Kahn, the program might look like this.

```
Process f(integer in X; integer out Y);
    Begin integer N, M;
        N := wait(X);
        Repeat
            Begin
                M := wait(X);
                send (N + M) on Y;
                N := M;
            End;
    End;
```

(The node starts by reading an integer, then it enters an infinite loop where it reads an integer and writes the sum of the two previous integers at each iteration.)

The corresponding function can be given by the following recursive definitions, where the function f corresponds to the node. This function takes a single argument corresponding to the input stream. The recursively defined function f' takes two arguments, the previous integer, i.e., the local state, and the input stream.

$$\begin{aligned}
&f(\epsilon) = \epsilon \\
&f(n.x) = f'(n, x) \\
&f'(n, \epsilon) = \epsilon \\
&f'(n, m.x) = (n + m).f'(m, x)
\end{aligned}$$

So, for example, $f(\epsilon) = \epsilon$. If we write $1.2.3.\epsilon$ for the string consisting of 1,2 and 3 we have

$$f(1.2.3.\epsilon) = f'(1, 2.3.\epsilon) = 3.f'(2, 3) = 3.5.f'(3, \epsilon) = 3.5.\epsilon$$

Now, suppose the input is extended to the string $1.2.3.4.\epsilon$. It is easy to see that $f(1.2.3.4.\epsilon) = 3.5.7.\epsilon$.

For comparison, we give the same program in the form of concurrent logic and concurrent constraint programs. First the clp program.

```
$F([N \mid X_1], Y) : -$
        $F'(N, X_1, Y).$
$F'(N, [M \mid X_1], Y) : -$
        $K$ is $N + M,$
        $Y = [K \mid Y_1],$
        $F'(M, X_1, Y_1).$
```

Next, the ccp program.

```
$F(X, Y) ::$
    $((\exists_N \exists_{X_1} X = [N|X_1]) \Rightarrow$
        $\exists_N \exists_{X_1} (X = [N|X_1] \wedge$
                $F'(N, X_1, Y)))$
$F'(N, X, Y) ::$
    $((\exists_M \exists_{X_1} X = [M|X_1]) \Rightarrow$
        $\exists_M \exists_{X_1} \exists_{Y_1} (X = [M|X_1] \wedge$
                $Y = [N + M \mid Y_1] \wedge$
                $F'(M, X_1, Y_1)))$
```

In the model of data flow described by Dennis [25], the program example we have been looking at can be assembled from the following primitive nodes. First, we need a node *duplicate* that reads input from a channel and sends it to its two output channels (Figure 6.1 (a)). Second, a node *butfirst* with one input and one output channel that transmits all input to the output channel, except the first token, which is discarded (Figure 6.1 (b)). Last, a node *add* with two input and one output channel which adds incoming tokens on the two input channels and outputs their sum (Figure 6.1 (c)). So if the input on the two channels is $1.2.3.4.\epsilon$ and $11.11.\epsilon$, the output is $12.13.\epsilon$. At this

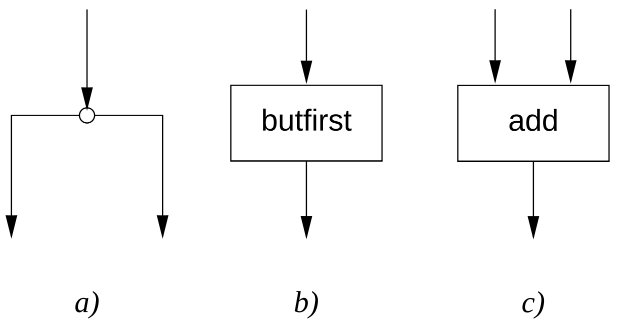

Figure 6.1: Three primitive nodes; *a*) duplicate, *b*) butfirst, and *c*) add.

stage, the adder is waiting for input from the second channel. When a token arrives, it will be added to 3 and the result sent to the output channel.

We can now use the primitives we have defined to compose the example program (Figure 6.2). The construction is fairly straight-forward. Use the duplication node to get two copies of the input channel, use the node *butfirst* to remove the first token from one of the channels, thus getting a sequence which is displaced one step. Finally the contents of the two channels are added.

6.2 Data flow networks

In this section we give an inductive definition of the set of data flow networks. The basic idea is that each data flow network has a finite set of input channels and output channels, and is composed using a small number of composition rules.

We will use $\alpha, \beta, \gamma, \dots$ for channels, I for sets of input channels, O for sets of output channels and K for arbitrary sets of channels. Let Σ be the set of tokens that may be sent over a channel, and $(D, \sqsubseteq)$ the cpo of finite and infinite strings over Σ, where $\sqsubseteq$ denotes the prefix ordering of strings.

6.2.1 Deterministic nodes

We assume that there is a set of deterministic nodes, and that for each deterministic node d with input channels I and output channels O there is a continuous function

$$f_d : D^I \to D^O$$

which gives the behaviour of the deterministic node. (We use the notation D^K for the lattice consisting of the set of functions $K \to D$, together with

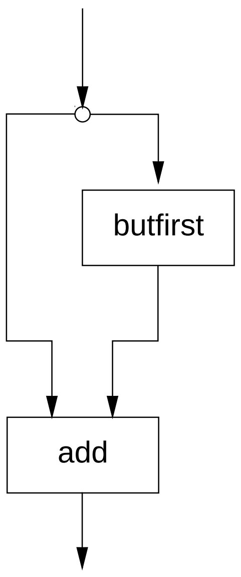

Figure 6.2: The example program as a dataflow network.

a top element.)

6.2.2 Non-deterministic nodes

To write non-deterministic data flow programs the only non-deterministic conctruct needed is the merge node. We will assume that the merge node has two input and one output channels and that it gives an angelic merge.

For example, given input streams

$$aaa\ldots \quad \text{and} \quad bb,$$

possible outputs from the merge node might be

$$bbaaa\ldots \quad \text{or} \quad abaabaaa\ldots \quad \text{or} \quad aaa\ldots.$$

6.2.3 Forming networks

A data flow network is a graph of nodes and communication channels. We will use two operations, **par** and **edge** in the construction of data flow networks.

Given networks F_1 and F_2,

$$F_1 \textbf{ par } F_2$$

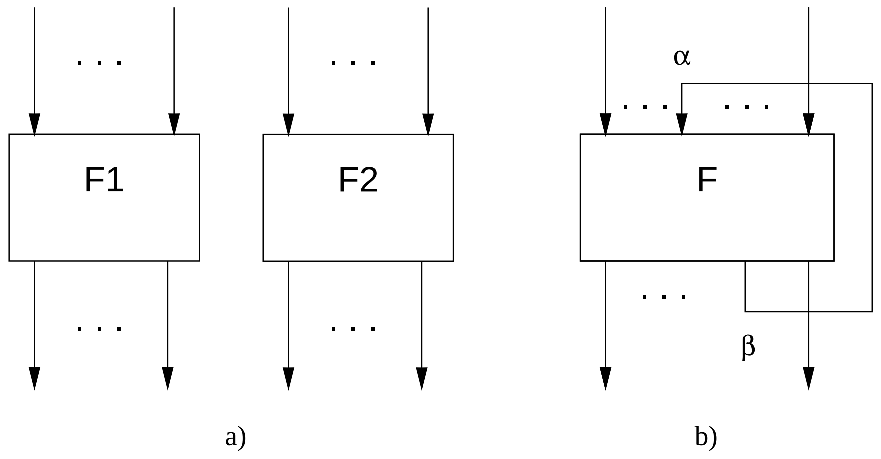

Figure 6.3: Ways to compose a network: *a*) parallel composition, *b*) adding edges.

creates a new network by putting the networks F_1 and F_2 together, without making any new connections (Figure 6.3 (*a*)). Suppose that F_1 is a network with input channels I_1 and output channels O_1, and F_2 a network with inputs I_2 and outputs O_2. Also assume that the sets I_1, I_2, O_1 and O_2 are all mutually disjoint. Now, F **par** F_2 creates a networks with input channels $I_1 \cup I_2$ and output channels $O_1 \cup O_2$, simply by putting the two networks next to each other. (Note that the **par** construct is only defined for networks with disjoint sets of input and output channels.)

The second construct adds an edge (a communication channel) to a given network. If F is a network with inputs I and outputs O, and $\alpha \in I$ and $\beta \in O$, the construction

$$\mathbf{edge}_{\alpha,\beta}F$$

gives a network where output channel β has been connected to input channel α (Figure 6.3 (*b*)). Thus, $\mathbf{edge}_{\alpha,\beta}F$ is a network with input channels $I \setminus \{\alpha\}$ and output channels $O \setminus \{\beta\}$.

It is easy to see that we can construct any network using **par** and **edge**. First, use **par** to create a network containing all desired nodes, and then use **edge** to add the edges.

6.2.4 The set of data flow programs

We give the set of data flow programs inductively, where $\mathrm{Net}_{I,O}$ is the set of nets with input channel I and output channels O.

Definition 6.2.1 For disjoint sets of channels I, O, let $\mathrm{Net}_{I,O}$ be the set of nets given by the following rules.

1. $d \in \mathrm{Net}_{I,O}$ if d is a deterministic node with inputs channels I and output channels O.
2. $\mathbf{merge} \in \mathrm{Net}_{\{\alpha,\beta\},\{\gamma\}}$, for distinct channels α, β and γ.
3. $F_1 \, \mathbf{par} \, F_2 \in \mathrm{Net}_{I_1 \cup I_2, O_1 \cup O_2}$, if $F_1 \in \mathrm{Net}_{I_1,O_1}$, and $F_2 \in \mathrm{Net}_{I_2,O_2}$.
4. $\mathbf{edge}_{\alpha,\beta} F \in \mathrm{Net}_{I \setminus \{\alpha\}, O \setminus \{\beta\}}$, if $\alpha \in I$, $\beta \in O$, and $F \in \mathrm{Net}_{I,O}$.

□

The definition of nets allow nets with an empty set of output channels. Any two nets with the same number of input channels and no output channels should have the the same abstract semantics since the two nets produce the same output, and there is no context in which the two nets can be distinguished.

6.3 Relating histories and constraints

To give a translation from data flow to ccp we must first specify a relationship between the the data structures of data flow nets (that is, histories) and the corresponding structures of ccp (constraints).

6.3.1 A constraint system of histories

Recall that Σ is the set of tokens that can appear on a channel in a data flow network. To represent histories of streams in a data flow network as constraints, we assume that Σ is the set of integers, and use the constraint system of integers and lists of integers described in Chapter 3.

Note that the constraints include limits of sequences of formulas, so there will, for example, be a constraint which is the limit of the sequence

$$\exists_Y X = [a_0 \mid Y] \quad \exists_Y X = [a_0, a_1 \mid Y] \quad \exists_Y X = [a_0, a_1, a_2 \mid Y] \quad \ldots$$

for $a_0, a_1, a_2, \ldots \in \Sigma$. Thus, the finite constraints we will consider in the translation will be constraints that map variables to finite strings. In the example above, the constraint which is the limit of the sequence would map X to the infinite string $a_0 a_1 a_2 \ldots$.

6.3.2 Correspondence between channels and variables

We assume that there is a one-to-one mapping from channels to variables. If α is an input channel and β an output channel, we will write X_α and Y_β for the corresponding variables. For a channel γ that may be either an input channel or an output channel we will write Z_γ for the corresponding variable.

6.3.3 Correspondence between histories and constraints

We define a mapping from histories $\sigma \in D^K$ to constraints. For $\sigma \in D^K$, where K is a set of variables, let

$$H\sigma = \bigsqcup\{\exists_Y(Z_\gamma = [a_0, a_1, \ldots, a_n \mid Y]) \mid \gamma \in K, \sigma\,\gamma = a_0 a_1 \ldots a_n\}$$

The corresponding mapping from constraints c to histories D^K is written $c \lceil K$, and is defined by

$$c \lceil K = \bigsqcup\{\sigma \mid H\,\sigma \sqsubseteq c\}.$$

In other words, $c \lceil K = \sigma$, where $\sigma \in D^K$ is the strongest such that whenever $\sigma(\gamma) \sqsupseteq a_0 a_1 \ldots a_n$, for $\gamma \in K$, we have

$$c \sqsupseteq (\exists_Y Z_\gamma = [a_0, a_1, \ldots, a_n \mid Y]).$$

It is easy to see that the pair of $H : D^K \to \mathcal{U}$ and $\cdot \lceil K : \mathcal{U} \to D^K$ is a Galois connection.

6.3.4 Correspondence between functions over histories and closure operators over constraints

Given disjoint sets I and O of channels, and a closure operator f over constraints, the corresponding function from histories over I to histories over O is written $f \lceil (I \to O)$ and is defined to be equal to g, where

$$g\,\sigma = \sqcup\{\rho \in D^O \mid f(H\,\sigma) \sqsupseteq H\,\rho\}.$$

If $f \in D^I \to D^O$, let $(\!(f)\!)$ be the corresponding closure operator over constraints, defined according to the equation

$$(\!(f)\!)c = c \sqcup (H(f\,(c \lceil I))).$$

It is easy to see that the pair of

$$(\!(\cdot)\!) : (D^I \to D^O) \to \mathcal{C} \qquad \text{and} \qquad \cdot \lceil (I \to O) : \mathcal{C} \to (D^I \to D^O)$$

is a Galois connection, where $\mathcal{C}$ is the lattice of closure operators over constraints.

6.4 Relationship to ccp

As we have already suggested, the language of data flow networks can be seen as a special case of concurrent constraint programming. In this section we give a translation from data flow networks to ccp agents.

A net F with input channels $I = \{\alpha, \ldots\}$ and output channels $O = \{\beta, \ldots\}$ is translated into an agent $A(X_\alpha, \ldots; Y_\beta, \ldots)$. We separate the arguments corresponding to input and output channels with a semicolon, for greater clarity. The variables $X_\alpha, \ldots$ will be referred to as *input variables*, and the variables Y_β as *output variables*.

For the rest of this chapter, we will assume a program Π that contains all necessary definitions of procedures corresponding to various primitive nodes of the language of data flow networks.

Deterministic nodes For a deterministic node $d \in \mathrm{Net}_{I,O}$, we assumed that there is a continuous function $f_d : D^I \to D^O$ which gives the behaviour of the node. Thus, we require that the functionality of each trace in the semantics of d must be weaker or equal to the function f_d. We must also require that the output of the trace to be equal to the result of applying f_d to the input of the trace. In other words, a deterministic node must eventually produce the result given by the function f_d.

For each node d we assume that there is a deterministic concurrent constraint procedure $\mathrm{node}_d(X_\alpha, \ldots, Y_\beta, \ldots)$, such that the corresponding closure operator is $(\!(f_d)\!)$.

Example 6.4.1 Consider a deterministic node, with one input channel α and one output channel β, which copies its input to the output channel. The behaviour of this node is given by the identity function. The corresponding closure operator is the weakest closure operator which maps a constraint in which X_α is bound to a string w to a constraint in which Y_β is bound to the same string w. □

Merge nodes We use the a simplified version of the merge procedure, which was defined in Section 3.11.

$$
\begin{array}{l}
\mathrm{merge}(X, Y, Z) :: \\
\quad (\exists_A \exists_{X_1} (X = [A \mid X_1]) \\
\qquad \Rightarrow \exists_A \exists_{X_1} \exists_{Z_1} (X = [A \mid X_1] \\
\qquad\qquad\qquad \wedge\, Z = [A \mid Z_1] \\
\qquad\qquad\qquad \wedge\, \mathrm{merge}(X_1, Y, Z_1)) \\
\quad [\,]\ \exists_A \exists_{Y_1} (Y = [A \mid Y_1]) \\
\qquad \Rightarrow \exists_A \exists_{Y_1} \exists_{Z_1} (Y = [A \mid Y_1] \\
\qquad\qquad\qquad \wedge\, Z = [A \mid Z_1] \\
\qquad\qquad\qquad \wedge\, \mathrm{merge}(X, Y_1, Z_1)))
\end{array}
$$

The difference between this merge procedure, and the one defined in Section 3.11 is that the cases that treat terminated lists have been removed.

A net **merge** $\in \text{Net}_{\{\alpha,\beta\},\{\gamma\}}$ is mapped to the agent $\text{merge}(X_\alpha, X_\beta; Y_\gamma)$.

Parallel composition Parallel composition of nets is mapped to a conjunction of agents. If $F_1 \in \text{Net}_{I_1,O_1}$ and $F_2 \in \text{Net}_{I_2,O_2}$, and F_1 is mapped to A_1 and F_2 to A_2, we can map F_1 **par** F_2 to $A_1 \wedge A_2$, assuming that I_1, I_2, O_1 and O_2 are all mutually disjoint.

Adding edges When we add an edge between an input channel and an output channel two things happen; first, the two channels are connected so that any data output on the output channel will appear on the input channel, second, the two channels are hidden and can not be accessed by any outside observer.

We could express the copying from the output channel β to the input channel α as a unification $X_\alpha = Y_\beta$, but we choose to make the copying explicit, by a procedure $\text{copy}(Y_\beta; X_\alpha)$ which is defined as follows.

$$\begin{aligned}
&\text{copy}(Y; X) :: \\
&\qquad (\exists_A \exists_{Y_1} (Y = [A \mid Y_1]) \\
&\qquad\qquad \Rightarrow \exists_A \exists_{Y_1} \exists_{X_1} (Y = [A \mid Y_1] \\
&\qquad\qquad\qquad\qquad \wedge\, X = [A \mid X_1] \\
&\qquad\qquad\qquad\qquad \wedge\, \text{copy}(Y_1; X_1)))
\end{aligned}$$

The second part of the edge construct, the hiding of the two channels, is of course accomplished with an existential quantifier.

If the net F is mapped to the agent

$$A,$$

the net $\mathbf{edge}_{\alpha,\beta} F$ is mapped to the agent

$$\exists_{X_\alpha} \exists_{Y_\beta} (A \wedge \text{copy}(Y_\beta, X_\alpha)).$$

6.5 Examining the results of the translation

The translation from data flow to ccp is compositional in the sense that the translation of a net constructor only depends on the translation of its components. For example, the translation of a net $\mathbf{edge}_{\alpha,\beta} F$ is given in terms of the translation of the net F. It follows that we have a compositional semantics for data flow which gives the meaning of a data flow net as a set of ccp traces.

An agent that is the result of a translation of a data flow network to ccp will be referred to as a *df-agent*. In this section we examine the properties of df-agents. In the next section, we will derive a compositional semantics for data flow nets.

Say that a *df-trace* is a trace such that

1. $v(t)_0 = H\sigma$, for some $\sigma \in D^I$.
2. When $i \in \omega \setminus r(t)$, we have $v(t)_{i+1} = v(t)_i \sqcup H\sigma$, for some $\sigma \in D^I$.
3. When $i \in r(t)$, we have $v(t)_{i+1} = v(t)_i \sqcup H\rho$, for some $\rho \in D^O$.

Thus, the df-traces are the traces whose input steps involve input variables only, and output steps involve output variables only.

The semantics of a df-agent may contain traces which are not df-traces, but each such trace can be derived from a df-trace in the following manner. Let t be a df-trace, and $(c_i)_{i\in\omega}$ be a chain of constraints which do not involve any input variables, and is such that $c_i = c_{i+1}$ for $i \in r(t)$. The sequence $(c_i)_{i\in\omega}$ is intended to represent input to the agent which affects variables other than the input variables. The trace t', given by $r(t') = r(t)$ and $v(t')_i = v(t)_i \sqcup c_i$ is also a trace of A. It is easy to see that each trace of a df-agent can be obtained in this manner. It follows that we can give the semantics of a df-agent in terms of its df-traces.

It turns out that when we give a semantics for df-agents, there are three properties of df-traces that are relevant.

1. The *input* received by the trace, which is simply the component of the limit of the trace which concerns the values of input variables.
2. The total *output* produced by the trace, this is the component of the limit of the trace which concerns the values of output variables.
3. The *directed functionality* of the trace, that is, the functionality of the trace when seen as a function from the values of input variables to the values of output variables.

Definition 6.5.1 Suppose that $F \in \mathrm{Net}_{I,O}$, that A is the correspoding df-agent, and t is a df-trace of A. We define the following operations on t.

1. Let $\mathrm{in}\, t = (\lim t) \lceil I$.
2. Let $\mathrm{out}\, t = (\lim t) \lceil O$.
3. Let $\mathrm{dfn}\, t = (\mathrm{fn}\, t) \lceil (I \to O)$.

□

Note that the operations defined above are given with respect to sets of channels I and O. When we apply these operations on a df-trace t, the sets I and O will always be given in the context; either given directly, or, in the case t is associated to a particular net F, assumed to be the sets of input and output channels of F.

6.6 Compositional semantics for data flow nets

We can now give the semantics of a data flow net by a translation to df-agents, and the the semantics of df-agents can then be given as a set of df-traces. Thus we obtain a semantics for dataflow nets. It follows immediately from the compositionalty of the abstract semantics for ccp that the corresponding semantics for data flow nets is compositional. In this section, we will give the semantic rules for composition of nets. For a net F, let $\mathcal{S}[\![F]\!]$ be the set of df-traces of the net.

6.6.1 Deterministic nodes

The df-traces of an agent $\mathrm{node}_d(X_\alpha, \ldots ; Y_\beta, \ldots)$ are the df-traces t with functionality weaker than or equal to the closure operator g_d, and limit which is a fixpoint of g_d, where $g_d = (\!(f_d)\!)$.

Thus, we find that for any df-trace t of d we have $\mathrm{dfn}\, t \sqsubseteq f_d$, and that $f_d(\mathrm{in}\, t) = \mathrm{out}\, t$.

Example 6.6.1 Consider a deterministic node with one input channel and one output channel. The corresponding procedure is as defined by the procedure copy defined in Section 6.4. One possible trace of a call $\mathrm{copy}(X; Y)$ is one where X is first bound to a list of three elements, which are then copied to Y, i.e., the trace t where

$$\begin{aligned}
v(t)_0 &= (\exists_{X_1} X = [1,2,3 \mid X_1]) \\
v(t)_1 &= (\exists_{X_1} X = [1,2,3 \mid X_1] \wedge \exists_{Y_1} Y = [1 \mid Y_1]) \\
v(t)_2 &= (\exists_{X_1} X = [1,2,3 \mid X_1] \wedge \exists_{Y_1} Y = [1,2 \mid Y_1]) \\
v(t)_3 &= (\exists_{X_1} X = [1,2,3 \mid X_1] \wedge \exists_{Y_1} Y = [1,2,3 \mid Y_1]) \\
v(t)_i &= (\exists_{X_1} X = [1,2,3 \mid X_1] \wedge \exists_{Y_1} Y = [1,2,3 \mid Y_1]), \quad \text{for } i \geq 4,
\end{aligned}$$

and $r(t) = \{0, 1, 2\}$.

We find that $\mathrm{in}\, t = \mathrm{out}\, t = 123$, and $\mathrm{dfn}\, t$ is the least continuous function $f : D \to D$ such that $f(123) \sqsupseteq 123$. □

6.6.2 Merge

The set of traces for the merge node follow directly from the definition of the fully abstract semantics of the merge procedure, but we want to give the semantics of data flow networks without reference to ccp programs.

Let an *oracle* be a finite or infinite sequence $s \in \{0,1\}^\infty$. For $n, m \in \{0, 1, 2, \ldots\} \cup \{\omega\}$ and an oracle s define

$$\mathrm{dmerge}(n, m, s) : D \times D \to D$$

according to the following rules. We will assume that $1 + \omega = \omega$.

$$\begin{aligned}
\text{dmerge}(1+n, m, 0.s)(a.x, y) &= a.\text{dmerge}(n, m, s) \\
\text{dmerge}(n, 1+m, 1.s)(x, b.y) &= b.\text{dmerge}(n, m, s) \\
\text{dmerge}(0, m, 0.s)(x, y) &= y \\
\text{dmerge}(n, 0, 1.s)(x, y) &= x \\
\text{dmerge}(n, m, s)(x, y) &= \epsilon, \qquad \text{if none of the rules above apply}
\end{aligned}$$

The idea is that given input histories x and y and some oracle s, evaluation of $\text{dmerge}(|x|, |y|, s)(x, y)$ should give one possible result of the merge node.

Example 6.6.2 Suppose $x = 4.5.\epsilon$, $y = 2.3.\epsilon$, and $s = 0.0.0.\ldots$, we have

$$\begin{aligned}
\text{dmerge}(|x|, |y|, s)(x, y) &= \text{dmerge}(2, 2, 00\ldots)(45, 23) \\
&= 4.\text{dmerge}(1, 2, 00\ldots)(5, 23) \\
&= 4.5.\text{dmerge}(0, 2, 00\ldots)(\epsilon, 23) \\
&= 4.5.2.3
\end{aligned}$$

If we now consider prefixes of x and y, for example $x' = 4.\epsilon$ and $y' = 2.\epsilon$, we find that

$$\text{dmerge}(|x|, |y|, s)(x', y') = 4.\epsilon,$$

which is a prefix of the string given by $\text{dmerge}(|x|, |y|, s)(x, y)$. It is easy to see that $\text{dmerge}(n, m, s)$ is continuous, for fixed n, m and s. □

Given input channels α_1 and α_2 and output channel β, and writing (x, y) for $\sigma \in D^{\{\alpha_1, \alpha_2\}}$ such that $\sigma\,\alpha_1 = x$ and $\sigma\,\alpha_2 = y$, the traces of a merge node can be given as the set

$$\begin{aligned}
\{t \mid\ & s \in \{0,1\}^\infty, \\
& \text{in}\, t = (x, y), \\
& \text{out}\, t = \text{dmerge}(|x|, |y|, s)(x, y), \text{ and} \\
& \text{dfn}\, t \sqsubseteq \text{dmerge}(|x|, |y|, s)\}
\end{aligned}$$

6.6.3 Parallel composition

Let $F \in \text{Net}_{I,O}$ such that $F = F_1 \,\mathbf{par}\, F_2$, and $F_1 \in \text{Net}_{I_1,O_1}$ and $F_2 \in \text{Net}_{I_2,O_2}$. Clearly, the df-agents corresponding to F_1 and F_2 cannot interact since their sets of free variables are disjoint. Thus, if we run the network F and only consider communications through the channels in I_1 and O_1, we will make the same observations as if we were running the network F_1.

Thus, the traces of F are the traces t such that there are traces $t_1 \in \mathcal{S}[\![F_1]\!]$ and $t_2 \in \mathcal{S}[\![F_2]\!]$ such that

1. $(\text{in}\, t) \restriction I_1 = \text{in}\, t_1$, $(\text{out}\, t) \restriction O_1 = \text{out}\, t_1$, and $(\text{dfn}\, t) \restriction I_1 \to O_1 = \text{dfn}\, t_1$, and

2. $(\mathrm{in}\, t) \upharpoonright I_2 = \mathrm{in}\, t_2$, $(\mathrm{out}\, t) \upharpoonright O_2 = \mathrm{out}\, t_2$, and $(\mathrm{dfn}\, t) \upharpoonright D^{I_2} = \mathrm{dfn}\, t_2$,

where $\upharpoonright$ is taken to be the usual restriction of a function to a subset of its domain.

6.6.4 Joining edges

The edge construct does two things; tokens output on channel β are copied onto channel α, and the two channels are hidden and cannot be accessed from the outside. We consider the traces of the net $F \in \mathrm{Net}_{I,O}$, where F is $\mathbf{edge}_{\alpha,\beta} F'$, for some net F'. We will first give an informal presentation of the rules for the dataflow semantics, and then relate these to the corresponding rules for ccp.

Let t' be a trace of the net F'. When is there a corresponding trace of F? Clearly, we must require that the total input on channel α is the same as the output on channel β, that is, $(\mathrm{in}\, t')\alpha = (\mathrm{out}\, t')\beta$. We assume that t' satisfies this condition.

Now we can give the functionality and input and output of the corresponding trace t of F. We obtain the input and output by hiding the components of $\mathrm{in}\, t'$ and $\mathrm{out}\, t'$ that give the history of channels α and β.

The functionality is a little bit more complicated. We assume that t' is a trace of F' which satisfies $(\mathrm{in}\, t')\alpha = (\mathrm{out}\, t')\beta$.

Note that finding the functionality of a trace of F actually involves a fixpoint computation, and this is most conveniently done if we work with closure operators instead of functions. So we convert the functionality of t', which is $\mathrm{dfn}\, t'$ into a closure operator and then compose it with a closure operator that describes the copying of information from channel β to channel α.

Let $\mathbf{path}(\beta, \alpha) : D^{I'} \times D^{O'} \to D^{I'} \times D^{O'}$ be defined according to

$$\mathbf{path}(\beta, \alpha)(\sigma, \rho) = (\sigma', \rho),$$

where $\sigma'\ \alpha = (\sigma\ \alpha) \sqcup (\rho\ \beta)$, and $\sigma'\ \gamma = \sigma\ \gamma$, for $\gamma \neq \alpha$. (In other words, $\mathbf{path}(\beta, \alpha)$ gives the behaviour of the procedure copy.)

All that is left to do now is hiding the channels α and β. Thus, the resulting functionality of some traces of F is

$$((\langle\!\langle \mathrm{dfn}\, t' \rangle\!\rangle \sqcap \mathbf{path}(\beta, \alpha)) \lceil (I \to O).$$

Thus t is a trace of $F \in \mathrm{Net}_{I,O}$, where $F = \mathbf{edge}_{\alpha,\beta} F'$ iff there is a trace t' of F' such that

1. $(\mathrm{in}\, t')\alpha = (\mathrm{out}\, t')\beta$,
2. $(\mathrm{in}\, t)\gamma = (\mathrm{in}\, t')\gamma$, for $\gamma \in I$,

$$\begin{aligned}
\mathcal{S}[\![d]\!] &= \{t \mid \mathrm{dfn}\, t \sqsubseteq f_d,\ \mathrm{out}\, t = f_d(\mathrm{in}\, t)\}\\
\mathcal{S}[\![\mathbf{merge}]\!] &= \{t \mid s \in \{0,1\}^\infty,\ \mathrm{in}\, t = (x,y),\\
&\qquad \mathrm{out}\, t = \mathrm{dmerge}(|x|,|y|,s)(x,y),\ \text{and}\\
&\qquad \mathrm{dfn}\, t \sqsubseteq \mathrm{dmerge}(|x|,|y|,s)\}\\
\mathcal{S}[\![F_1\, \mathbf{par}\, F_2]\!] &= \{t \mid t_1 \in \mathcal{S}[\![F_1]\!],\ t_2 \in \mathcal{S}[\![F_2]\!],\\
&\qquad (\mathrm{in}\, t) \restriction I_1 = \mathrm{in}\, t_1, (\mathrm{in}\, t) \restriction I_2 = \mathrm{in}\, t_2,\\
&\qquad (\mathrm{out}\, t) \restriction O_1 = \mathrm{out}\, t_1, (\mathrm{out}\, t) \restriction O_2 = \mathrm{out}\, t_2,\\
&\qquad (\mathrm{dfn}\, t) \restriction D^{I_1} = \mathrm{dfn}\, t_1, (\mathrm{dfn}\, t) \restriction D^{I_2} = \mathrm{dfn}\, t_2\}\\
\mathcal{S}[\![\mathbf{edge}_{\alpha,\beta} F]\!] &= \{t \mid t' \in \mathcal{S}[\![F']\!],\ (\mathrm{in}\, t')\alpha = (\mathrm{out}\, t')\beta,\\
&\qquad (\mathrm{in}\, t)\gamma = (\mathrm{in}\, t')\gamma,\ \text{for } \gamma \in I,\\
&\qquad (\mathrm{out}\, t)\gamma = (\mathrm{out}\, t')\gamma,\ \text{for } \gamma \in I,\ \text{and}\\
&\qquad \mathrm{dfn}\, t = (((\mathrm{dfn}\, t')) \cap \mathbf{path}(\beta,\alpha)) \lceil (I \to O)\}
\end{aligned}$$

Figure 6.4: The compositional semantics for data flow nets

3. $(\mathrm{out}\, t)\gamma = (\mathrm{out}\, t')\gamma$, for $\gamma \in O$, and
4. $\mathrm{dfn}\, t = (((\mathrm{dfn}\, t')) \cap \mathbf{path}(\beta,\alpha)) \lceil (I \to O)$.

6.6.5 Summary of the compositional semantics

We summarize the compositional semantics for data flow networks in Figure 6.4.

6.7 Full abstraction

Recall that a compositional semantics is considered to be fully abstract if it contains no redundant information, i.e., if the semantics of two program fragments differ, there should be some context for which the difference of behaviour of the two program fragments result in a difference in the behaviour of the whole program. In this section we will briefly recall the proof(s) of full abstraction for ccp given in Sections 5.4 and 5.6 and sketch the corresponding proofs for data flow nets.

When we gave a fully abstract semantics for ccp, the result semantics gave the behaviour of a program when run non-interactively, that is, the program received input data at the beginning of execution, and then no further input was given. The result semantics of a df-agent gives in the same way the behaviour of the corresponding data flow network when run non-interactively.

Both proofs of full abstraction for ccp went as follows. Assume that the two agents under consideration (A_1 and A_2) have different semantics. This implies that there is a trace t which belongs to the semantics of one agent

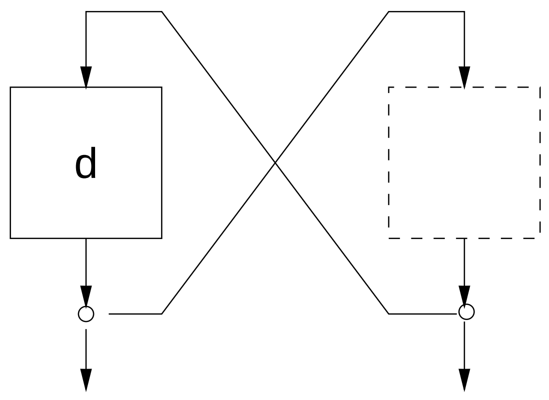

Figure 6.5: The context $C[\cdot]$.

(say A_1) but not the other (A_2). Construct an agent B which somehow generates an 'inverse' trace of t, that is, a trace $\bar{t}$ which follows the same sequence of stores as t, but performs input steps when t performs computation steps, and vice versa. The context is now a conjunction $B \wedge \cdot$, and it follows from the semantics of ccp that result semantics of the agents $B \wedge A_1$ and $B \wedge A_2$ should differ in that the first agent can produce output $\lim t$ when run without input, while the other cannot.

What should the corresponding proof for data flow networks look like? Given data flow networks $F_1, F_1 \in \mathrm{Net}_{I,O}$ we can construct a context as follows. View $\bar{t}$ as a trace of a net in which the input channels are O and the output channels are I. Assume that there is a node $d \in \mathrm{Net}_{O,I}$ such that $f_d = \mathrm{dfn}\, \bar{t}$, and construct a context $C[\cdot]$ as indicated in Figure 6.5.

It should be clear that when the network $C[F_1]$ is run (without input), it is possible to get as a result the pair $(\mathrm{in}\, t, \mathrm{out}\, t)$. Is the same result possible in the network $C[F_2]$? Suppose it is. If follows that F_2 has a trace t' with $\mathrm{in}\, t' = \mathrm{in}\, t$ and $\mathrm{out}\, t' = \mathrm{out}\, t$. However, it is easy to see that t' must have a functionality which is at least as strong as the one of t, which leads us to a contradiction.

The proof sketched above is not completely satisfactory, since we assumed that for each continuous function there was some deterministic node that computed the function, and this is obviously a very optimistic assumption! As an alternative we can employ a technique which we have already used in Section 5.6. This technique was first described by Russell [68], and the idea is that we construct a net that reads a representation of a trace and emulates a deterministic node whose functionality is the functionalty given by the trace (Figure 6.6). It is a straight-forward matter to find an

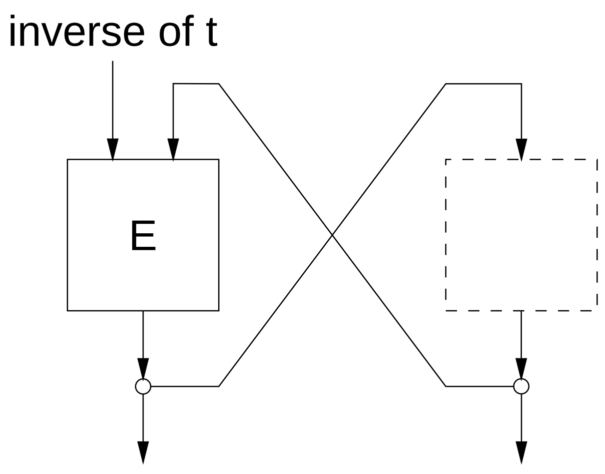

Figure 6.6: In this context, the node E reads a representation t of a function and emulates it.

appropriate representation of t which allows a representation of the trace to be sent over a finite set of channels.

6.8 Conclusions

This chapter described how the techniques that had been developed to give a fully abstract semantics for concurrent constraint programming languages also could be applied to a data flow language.

Since all fully abstract semantics for a programming language are, in a sense, isomorphic, it follows that the semantics presented here must be equivalent to the ones presented by Kok [40] and Jonsson [35]. However, the formulation here involves only two aspects of a trace, the limit of its input and output, and its functionality. In contrast, Kok's and Jonsson's models explicitly treats traces as sequences of communication events. Thus, one can argue that the semantic rules for composition in the model given here are more abstract, in that they involve fewer aspects of a trace.

The semantics presented here is also more general in the sense that it can also be applied on non-deterministic applicative languages with complex data types.

Chapter 7

There is no Fully Abstract Fixpoint Semantics for Non-deterministic Languages with Infinite Computations

It is well-known that for many non-deterministic programming languages it is not possible to give a fully abstract fixpoint semantics. This is usually attributed to "problems with continuity", that is, the assumption that the semantic functions should be continuous supposedly plays a role in the difficulties of giving a fully abstract fixpoint semantics. In this chapter, we show that for a large class of non-deterministic programming languages it is not possible to give a fully abstract least fixpoint semantics even if one considers arbitrary functions (not necessarily continuous) over arbitrary partial orders (not necessarily complete). It should also be noted that the negative result can easily be generalised to handle other types of fixpoint semantics, for example, fixpoints semantics based on category theory.

7.1 Assumptions

We consider a minimal programming language which satisfies the following properties.

1. There is some form of *non-deterministic choice*.
2. The program can generate *output*,
3. The language allows arbitrary recursion, i.e., non-guarded recursion is allowed.
4. The results of *infinite computations* are considered.

It turns out that the class of context-free grammars provide a simple model which satisfies almost all listed requirements. A context-free grammar has a set of terminal symbols, which correspond to output actions, the non-terminal symbols correspond to procedure calls, for each non-terminal we can have any number of productions, giving a non-deterministic choice, and each right-hand side of a production is a sequence of terminal and non-terminal symbols, giving a sequential composition. The standard definition of the set of words generated by a grammar does not allow infinite derivations, but it is possible to extend the derivations to allow also words generated by infinite derivations. We thus obtain a simple programming language with the notation of context-free grammars, and an operational semantics which is close to the standard rules of context-free grammars.

7.2 Related Work

It is well known that many non-deterministic languages do not have a continuous fully abstract fixpoint semantics. Abramsky [2] considered a simple non-deterministic language similar to the one examined in this and showed that there could be no *continuous* fully abstract fixpoint semantics.

Apt and Plotkin [4] considered an imperative programming language with unbounded non-determinism and while-loops and showed that there could be no *continuous* fully abstract fixpoint semantics. However, they were able to give a fully abstract least fixpoint semantics by giving up the requirement that the semantic functions should be continuous. To see how this is possible, note that by the Knaster-Tarski theorem any monotone function over a complete lattice has a least fixed point. In other words, to define a fixpoint semantics it is sufficient to have semantic functions that are monotone. This approach has been applied in some recent papers, see for example Barrett [6]. A related approach was presented by Roscoe [67].

The main differences between the language we consider, and the one Apt and Plotkin examined are that our language allows arbitrary recursion and infinite computations, but not unbounded non-determinism. The language in this paper is also simpler since it is not a regular programming language and has no state or value-passing etc.

There are many examples of denotational semantics for languages which satisfy three of the four properties listed above.

For example, Kahn's semantics [37] treats infinite computation and arbitrary recursion but does not allow non-determinism.

Brookes [13] gives a fully abstract fixpoint semantics of an imperative non-deterministic language with shared variables. The semantic model also allows infinite traces and is thus able to adequately model the behaviour of infinitely running processes. However, recursion is not dealt with.

Saraswat, Rinard and Panangaden [71] give fully abstract fixpoint semantics for various types of concurrent constraint programming languages.

One of the languages is a non-deterministic language which allows arbitrary recursion. However, only finite computations are considered.

A similar result is by Russel [69], who considers a class of non-deterministic data flow networks. He gives a fully abstract fixpoint semantics but does not consider infinite computations.

The language we consider is based on context-free grammars. The difference is mainly that we consider strings generated by infinite left-most derivations. This very simple model of non-deterministic computation has previously been studied by Nivat [56] and Poigné [66].

7.3 Relevance and significance

Why look at the semantics of non-deterministic and non-terminating programs? These programs have a simple operational behaviour, and one would expect the same to hold for their fixpoint semantics. Moreover, there are many programs that can respond to input from more than one source, and which do not terminate unless the user asks the program to terminate. These programs are non-deterministic, if we do not include timing in the semantic model, and are potentially non-terminating.

Why is full abstraction important? One of the strong points of denotational semantics is that a denotational semantics of a programming language provides a mathematical structure that is in direct correspondence with the 'meaning' of expressions in the program. The structure contains precisely that information which is relevant to understand the behaviour of expressions within various contexts. If one gives a denotational semantics which is not fully abstract, this correspondence is lost, and thus one of the reasons for giving a denotational semantics.

7.4 A simple language

As mentioned in the introductory section, we will base our formalism on the context-free grammars. To obtain a suitable operational semantics we will extend the set of derivations of a grammar to also allow infinite derivations.

The following presentation is based on Cohen and Gold [20].

7.4.1 Generation of infinite words

Let $\mathcal{N}$ be an infinite set of *non-terminals* and $\mathcal{T}$ be an infinite set of *terminals*. A *grammar* is then a finite set of productions of the form $X \to \alpha$, where X is a non-terminal, and α is a finite string over $\mathcal{T} \cup \mathcal{N}$. A finite string over $\mathcal{T} \cup \mathcal{N}$ will sometimes be referred to as an *agent*. A *generalised agent* is a an agent or a finite set of agents. For a grammar G and a non-terminal

X, let $G(X)$ be the set of agents α such that there is a production $X \to \alpha$ in G (thus $G(X)$ is a generalised agent).

For a grammar G we define the relation $\overset{G}{\Longrightarrow}$ to be the smallest relation over $(\mathcal{T} \cup \mathcal{N})^*$ that satisfies

$$uX\alpha \overset{G}{\Longrightarrow} u\beta\alpha,$$

for a string $u \in \mathcal{T}^*$, a non-terminal X and strings $\alpha, \beta \in (\mathcal{T} \cup \mathcal{N})^*$ such that there is a production $X \to \beta$ in G. Let $\overset{G}{\Longrightarrow}^*$ be the transitive and reflexive closure of $\overset{G}{\Longrightarrow}$. (Whenever the grammar is given in the context, we will omit the index G.)

We will not require the grammar to have a particular start symbol, since we want to be able to reason about languages generated from different words.

First, we consider the case where a finite word is generated by a derivation that terminates after a finite number of steps. The language generated by a grammar G and an agent α is

$$\mathcal{L}(G, \alpha) = \{w \in \mathcal{T}^* \mid \alpha \Longrightarrow^* w\}.$$

When an infinite derivation $\alpha_0 \Longrightarrow \alpha_1 \Longrightarrow \ldots$ is considered one can imagine many different definitions of which word is generated. We want to see a grammar as a sequential program where a terminal symbol would correspond to an atomic action. This operational view suggests the following definition, which is by Cohen and Gold.

For a string $\alpha \in (\mathcal{T} \bigcup \mathcal{N})^*$, say that w is the *largest terminal prefix* of α if $w \in \mathcal{T}^*$, and w is a prefix of α, and every word $w' \in \mathcal{T}^*$ which is a prefix of α is also a prefix of w.

Given an infinite derivation $\alpha_0 \longrightarrow \alpha_1 \ldots \alpha_n \ldots$ we can construct a chain $w_0 \leq w_1 \leq \ldots w_n \leq \ldots$ such that w_i is the largest terminal prefix of α_i, for all $i \in \omega$. We say that $w = \bigsqcup_{i \in \omega} w_i$ is the generated word and use the notation $\alpha_0 \Longrightarrow^\omega w$. Note that w can be finite or infinite.

The *ω-language* generated by the agent α and the grammar G is

$$\mathcal{L}^\omega(G, \alpha) = \{w \in \mathcal{T}^\omega \mid \alpha \Longrightarrow^\omega w\}.$$

For the set of words generated by finite and infinite derivations we write

$$\mathcal{L}^\infty(G, \alpha) = \mathcal{L}(G, \alpha) \cup \mathcal{L}^\omega(G, \alpha).$$

We generalise the definition to generalised agents α by

$$\mathcal{L}^\infty(G, \alpha) = \bigcup_{\beta \in \alpha} \mathcal{L}^\infty(G, \beta).$$

7.4.2 External behaviour

We have arrived at a fairly simple definition of the set of generated words (or, if you like, an operational semantics giving the set of sequences of actions performed by an agent). For a particular execution, the external behaviour consists of all output that will eventually be produced by the program, and nothing else. Note that it is possible to determine whether an agent α terminates by considering the set of strings generated from the agent αd, where d in a non-terminal not occurring in the grammar or in α. We argue that the set of strings generated by an agent corresponds exactly to the external behaviour of an agent.

7.5 There is no fully abstract fixpoint semantics

In this section we define which properties a fully abstract fixpoint semantics for our language should have, and prove that there cannot be a fixpoint semantics with these properties.

Typically, the domain of a denotational semantics is a complete lattice or a cpo, and the semantic functions are continuous. To make the definition of fixpoint semantics as general as possible, we will only make two assumptions about the properties of the domain and the semantic functions; that the semantic function for a grammar has a least fixpoint, and that the domain is a partial order (otherwise it is meaningless to speak about least fixpoints).

Definition 7.5.1 A *fixpoint semantics* consists of a partially ordered set D together with the following functions, for generalised agents α, and grammars G,

$$\mathcal{E}[\![\alpha]\!] : (\mathcal{N} \to D) \to D$$
$$\mathcal{P}[\![G]\!] : (\mathcal{N} \to D) \to (\mathcal{N} \to D)$$

such that the following folds.

1. (Fixpoints) For each grammar G, the function $\mathcal{P}[\![G]\!]$ has a least fixpoint.

2. (Correctness) Suppose that σ is the least fixpoint of $\mathcal{P}[\![G]\!]$ and σ' is the least fixpoint of $\mathcal{P}[\![G']\!]$. Whenever $\mathcal{E}[\![\alpha]\!]\sigma = \mathcal{E}[\![\alpha']\!]\sigma'$ it follows that $\mathcal{L}^\infty(G, \alpha) = \mathcal{L}^\infty(G', \alpha')$.

3. (Compositionality) For a grammar G, an environment σ, and a non-terminal X we have $\mathcal{P}[\![G]\!]\sigma X = \mathcal{E}[\![\alpha]\!]\sigma$, where $\alpha = G(X)$.

□

Motivation The idea is that the meaning of an agent α, with respect to a grammar G, should be given by $\mathcal{E}[\![\alpha]\!]\sigma$, where σ is the environment given by the least fixpoint of $\mathcal{P}[\![G]\!]$ (by the fixpoint condition we knows that a least fixpoint exists). The correctness condition says that the semantics should be able to predict the set of strings generated by an agent. This is of course a very natural requirement for any semantic model.

If a semantics is compositional we expect that the denotation of an expression should depend only of the denotations of its components. It follows from the compositionality condition that given functions $\mathcal{E}[\![\alpha]\!]$, the function $\mathcal{P}[\![G]\!]$ is uniquely defined. Note that the compositionality requirement implies that for grammars G, G' and non-terminals X, X' such that $G(X) = G'(X')$ we have $\mathcal{P}[\![G]\!]\sigma X = \mathcal{E}[\![G(X)]\!]\sigma = \mathcal{E}[\![G'(X')]\!]\sigma = \mathcal{P}[\![G']\!]\sigma X'$, for any environment σ.

The standard definition of full abstraction is that for any program constructs A and A' which are mapped to different elements in the semantic domain there should be a context $C[\cdot]$ such that $C[A]$ and $C[A']$ have different behaviour, the idea is that A and A' may have the same behaviour in themselves but when we put them in a context there may be a difference in behaviour.

In our language, there are no explicit operations for communication or change of state, so one might suspect that the compositionality requirement is unnecessary. However, consider as an example the agents α_1 and α_2, where the agent α_1 generates the empty string and terminates and α_2 generates the empty string without terminating. The difference between these two agents is detectable if we put them in the context

$$C[\cdot] = \cdot\, d.$$

The agent $\alpha_1 d$ will generate the string d while the agent $\alpha_2 d$ will only generate the empty string.

In general, if we have two agents α_1 and α_2, and a terminal symbol d that does not occur in any string generated by either α_1 or α_2, it is easy to see that if we know that the sets of strings generated by $\alpha_1 d$ and $\alpha_2 d$ are the same then it follows that in any context $C[\cdot]$ the sets of strings generated by $C[\alpha_1]$ and $C[\alpha_2]$ are identical. It follows that in the definition of full abstraction we only need to consider very simple contexts.

Definition 7.5.2 A fixpoint semantics is *fully abstract* if, for any given grammars G and G', a terminal symbol d that does not occur in either grammar, and agents α and α' such that $\mathcal{L}^\infty(G, \alpha d) = \mathcal{L}^\infty(G', \alpha' d)$, we have $\mathcal{E}[\![\alpha]\!]\sigma = \mathcal{E}[\![\alpha']\!]\sigma'$, where $\sigma = \text{fix}(\mathcal{P}[\![G]\!])$ and $\sigma' = \text{fix}(\mathcal{P}[\![G']\!])$. □

We are now ready for the result of this chapter.

Theorem 7.5.3 There is no fully abstract least fixpoint semantics.

Before we turn to the proof, the reader is asked to study the following two grammars.

Grammar G_1:

$$\begin{aligned}&A \rightarrow aA \mid \epsilon \mid D\\&B \rightarrow Aa \mid \epsilon \mid AD\\&D \rightarrow D\end{aligned}$$

Grammar G_2:

$$\begin{aligned}&A \rightarrow Aa \mid \epsilon \mid AD\\&B \rightarrow aA \mid \epsilon \mid D\\&D \rightarrow D\end{aligned}$$

We can assume that d is a terminal symbol (which of course does not occur in either G_1 or G_2). The grammars are variations of the ones given in the discussion in the preceding section, the difference is basically that among the set of strings generated from either A or B in either grammar will be finite strings $a\ldots$ generated by non-terminating computations.

It is easy to verify that the languages generated by grammars G_1 and G_2 from the agents Ad and Bd are the ones given by the following equations.

$$\begin{aligned}\mathcal{L}^\infty(G_1, Ad) &= a^*d \cup a^* \cup a^\omega\\\mathcal{L}^\infty(G_1, Bd) &= a^*d \cup a^* \cup a^\omega\\\mathcal{L}^\infty(G_2, Ad) &= a^*d \cup a^*\\\mathcal{L}^\infty(G_2, Bd) &= a^*d \cup a^*\end{aligned}$$

Note that $\mathcal{L}^\infty(G_1, Ad)$ and $\mathcal{L}^\infty(G_2, Ad)$ only differ in that the infinite string a^ω can be generated from grammar G_1. We are now ready to prove the theorem.

Proof. The proof is by contradiction. Suppose that there is a fully abstract fixpoint semantics. Let $f = \mathcal{P}[\![G_1]\!]$ and $g = \mathcal{P}[\![G_2]\!]$. Let σ_1 be the least fixpoint of the function f and σ_2 the least fixpoint of g. Since $\mathcal{L}^\infty(G_1, Ad) = \mathcal{L}^\infty(G_1, Bd)$ we can conclude that $f\sigma_1 A = f\sigma_1 B$, by the assumption that the semantics is fully abstract. It a similar way we can conclude from $\mathcal{L}^\infty(G_2, Ad) = \mathcal{L}^\infty(G_2, Bd)$ that $g\sigma_2 A = g\sigma_2 B$.

But from the compositionality requirement and the syntactic form of the grammars follows that for any $\rho \in (\mathcal{N} \rightarrow D)$ we have $f\rho A = g\rho B$ and $f\rho B = g\rho A$.

Now we can prove that the least fixpoint of f also is a fixpoint of g, and vice versa. Let $\sigma_1' = g\sigma_1$. We will prove that $\sigma_1' X = \sigma_1 X$, for any non-terminal. For the non-terminal A, we have

$$\begin{aligned}\sigma_1' A &= g\sigma_1 A && \text{(Definition of } \sigma_1'\text{)}\\&= f\sigma_1 B && \text{(Compositionality argument above)}\\&= f\sigma_1 A && \text{(By full abstraction)}\\&= \sigma_1 A && \text{(Since } \sigma_1 \text{ is a fixpoint of } f\text{)}\end{aligned}$$

For the non-terminal B, we have

$$\begin{aligned}\sigma_1'B &= g\sigma_1 B \\ &= f\sigma_1 A && \text{(By compositionality)} \\ &= f\sigma_1 B && \text{(By full abstraction)} \\ &= \sigma_1 B\end{aligned}$$

For the non-terminal D we have $\sigma_1'D = g\sigma_1 D = f\sigma_1 D = \sigma_1 D$. From this follows that $\sigma_1' = \sigma_1$.

We have shown that the least fixpoint of f also is a fixpoint of g. By a symmetric argument we can show that the least fixpoint of g also is a fixpoint of f. From this follows that f and g have the same least fixpoints, i.e., $\sigma_1 = \sigma_2$. But then $\mathcal{P}[\![G_1]\!]\sigma_1 A = \mathcal{P}[\![G_2]\!]\sigma_2 A$, which contradicts the observation that $\mathcal{L}^\infty(G_1, Ad) \neq \mathcal{L}^\infty(G_2, Ad)$. □

7.6 Discussion

If we are willing to give up full abstraction it is straight-forward to give a continuous fixpoint semantics. The easiest (and least interesting) way to give a fixpoint semantics is to base the semantic domain on the syntactic structure of the programming language. In our case the elements of the domain will be infinite (ordered) trees, where the internal nodes are labelled with either or (choice) or seq (sequence), and the leaves are labelled with a string of non-terminals or $\perp$. Given trees T_1 and T_2, say that $T_1 \sqsubseteq T_2$ if

1. T_1 consists of a leaf labelled $\perp$, or
2. if T_1 and T_2 are leaves so that the label of T_1 is a prefix of the label of T_2, or
3. if the root nodes of T_1 and T_2 have the same label, the same number of subtrees, and when $U_1, \ldots, U_n$ are the subtrees of the root of T_1 and $V_1, \ldots, V_n$ the subtrees of T_2 we have $U_k \sqsubseteq V_k$, for $k \leq n$.

Given this domain construction, it is straight-forward to define the appropriate semantic functions. Now, what information have we added to make the construction of a fixpoint semantics possible? First, the non-deterministic choices are now explicitly represented in the domain. Second, the choices are ordered, so when we compare two trees we can locate the respective descendants of the two trees which come from the same sequence of non-deterministic choices. As an example, consider the trees in Figure 7.1.

If we try to make the domain more abstract by making the trees unordered, and ignoring duplicated subtrees, i.e., view the subtrees of a node as a set, we see that it is no longer possible to define a reasonable partial order.

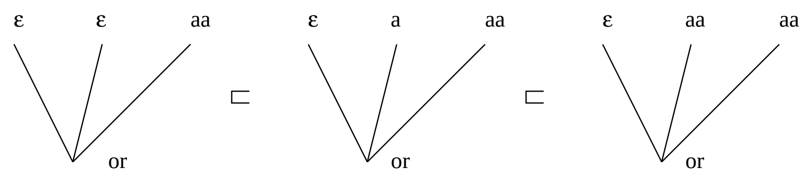

Figure 7.1: Representing non-deterministic behaviour as a tree of alternatives.

For example, given strings ϵ, a, and aa we have on one hand

$$\{\epsilon, \epsilon, aa\} \sqsubset \{\epsilon, a, aa\} \sqsubset \{\epsilon, aa, aa\}$$

but

$$\{\epsilon, \epsilon, aa\} = \{\epsilon, aa, aa\} = \{\epsilon, aa\}$$

It follows that the construction is not a partial order.

It appears that for any fixpoint semantics for a non-deterministic language with recursion and infinite computation the tree of non-deterministic choices must in some way be present, and when comparing different elements of the domain it must be possible to determine for each element the outcome of a particular sequence of choices.

7.7 Application to category-theoretic domains

As an alternative to the use of semantic domains based on various partial orders, such as complete lattices and cpo's, Lehmann [46, 47] proposed a class of categories called ω-categories. An ω-category is a category which has an initial object and where ω-colimits exist. An ω-functor is a functor which is continuous with respect to ω-colimits. It follows that each ω-functor has an initial fixpoint. Note that each cpo also is an ω-category, and each continuous function over a cpo can be seen as an ω-functor over the corresponding category.

Lehmann gave a powerdomain construction for ω-categories in which each set can be represented. However, the construction is based on multisets, which means that there may be many non-isomorphic representations of the same set. The powerdomain construction has previously been used by Abramsky [2], Panangaden and Russell [61], and Nyström and Jonsson [58] to model various forms of indeterminacy.

It is straight-forward to adapt the proof of Theorem 7.5.3 to show that there can be no fully abstract fixpoint semantics based on ω-categories for our language. First we must ask what it means for a semantics to be fully abstract when the semantic domain is a category. The natural choice is to require that agents with the same operational meaning should be mapped to isomorphic objects in the category. If we construct functors F and G analogous to the functions f and g in the proof of Theorem 7.5.3 we find that the initial fixpoint of F is a fixpoint of G, and vice versa. But this implies that the initial fixpoints of F and G are the same, which contradicts the correctness requirement.

We can thus see that a fixpoint semantics for a non-deterministic programming language with recursion and infinite computations which uses Lehmann's powerdomain construction can not be fully abstract in the sense that programs that have similar behaviour in all contexts have isomorphic semantics. For example, Abramsky notes ([2], page 4) that his category-theoretic semantics can not be fully abstract.

7.8 Conclusions

We have generalised the negative results of Abramsky [2] concerning fully abstract fixpoint semantics for non-deterministic languages to hold for a wide range of semantic models, in particular to non-continuous fixpoint semantics over partial orders and semantics based on continuous functions over categories. In contrast, the results of Apt and Plotkin [4] regarding a type of imperative programming languages with unbounded nondeterminism (that is, a nondeterministic assignment statement that can assign a variable any positive integer) ruled out any continuous semantics but for this type of language it was still possible to give a fully abstract non-continuous fixpoint semantics.

We can conclude that it does not help if one is willing to consider semantic domains that are not cpo's of semantic functions that are not continuous. It appears that any fixpoint semantics must maintain an explicit tree of all choices made.

The negative results of this paper can be immediately applied to a wide range of concurrent programming languages. The grammars used in the proof can be translated to concurrent constraint logic programs using the a transformation similar to the one used in DCG's. (This translation would map an agent which produces a string to a ccp agent which produces the corresponding list of terminal symbols.) Translation to a data flow language with recursion and non-determinism is also straight-forward.

Chapter 8

Oracle Semantics

In this chapter, we present an operational semantic of concurrent constraint programming based on *oracles*. An oracle is a sequence of integers, representing the non-deterministic choices made by an agent. Now, it is easy to see that many non-deterministic choices can only be made under certain conditions, for example, the non-deterministic choices in a selection depend on that the corresponding ask constraints are entailed by the store. The fact that each branch in a computation depends on conditions on the store implies that these conditions must be recorded in some way. The approach taken here is to record the conditions in the form of a *window*, which is a set containing the set of possible final outcomes 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 the use of oracles allows the non-deterministic behaviour of an agent to be isolated, it follows that we can show some confluence properties. The standard, finite, confluence property holds, and also a generalised confluence property concerning infinite sets of infinite computations.

The intention is that the oracle semantics should serve as a basis for a fixpoint semantics. In the the following chapter we will give two fixpoint semantics, one based on partial orders, and one based on category-theoretic domains. Since an agent with a given oracle is essentially deterministic it follows that we can use the techniques described by Saraswat, Rinard and Panangaden [71] 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.

8.1 Related Work

Many authors have used oracles to give the semantics of non-deterministic concurrent languages. For example, Cadiou and Levy [15] gave the operational semantics of a parallel imperative language in which the scheduling of processes was determined by an oracle. Milner [53] gave an operational model of a non-deterministic language in which oracles were used to determine non-deterministic choices. Keller [39], Kearney and Staples [38] and Russel [69] 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.

Marriott and Odersky [49] gave a confluence result of a concurrent constraint programming language in which the syntax had been extended to allow a representation of the branching structure of non-deterministic programs. Their result corresponds to Lemma 8.7.4.

8.2 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 essentially deterministic and can be seen as a closure operator over the domain of constraints, in the manner described by Saraswat et al. [71].

However, this is not sufficient. Consider an agent

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

The agent cannot select the first branch unless we know that the constraint $X = 1$ will hold eventually (and similarly that $Y = 2$ will hold eventually in the second branch). It is also possible for the agent to be suspended without ever selecting a branch. Fairness only allows the selection to be suspended indefinitely if none of the two constraints ever become entailed by the store. 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 *window*. The window gives one set of condition which must hold *eventually*, and one set of conditions which may *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 *not* be entailed by the store".

For example, for the agent

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

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 $X = 1$. The window is the condition that $X = 1$ and $Z = 3$ must hold eventually.

2. The second branch is analogous.

3. For the third branch, the 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.

8.3 Oracles

An *oracle* is a finite or infinite string over ω. Let 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, 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

$$\begin{aligned} \text{EVEN}(s) &= k_0 k_2 k_4 \ldots \\ \text{ODD}(s) &= k_1 k_3 k_5 \ldots, \end{aligned}$$

for $s = k_0 k_1 k_2 k_3 \ldots$.

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

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

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 $\pi_n s = s_n$, for $n \in \omega$.

For $\{s_n\}_{n\in\omega}$, let $\odot_{n\in\omega} s_i = s$ such that $\pi_n = s_n$, for $n \in \omega$.

8.4 Operational semantics

The oracle semantics is based on the idea that the non-deterministic decisions of an agent are controlled by an oracle. It is necessary to modify the operational semantics accordingly. We begin by extending the configurations to include oracles.

1. $c(s) : d \longrightarrow c(s) : c \sqcup d$

2. $$\frac{A^k(\pi_k s) : c \longrightarrow B^k(\pi_k s') : d, \quad k \in I}{\bigwedge_{j \in I} A^j(s) : c \longrightarrow \bigwedge_{j \in I} B^j(s') : d},$$
   where $B^j = A^j$ and $(\pi_j s') = (\pi_j s)$, for $j \in I \setminus \{k\}$.

3. $$\frac{c_k \sqsubseteq c, \quad 1 \le k \le n}{(c_1 \Rightarrow A_1 \,[]\, \ldots \,[]\, c_n \Rightarrow A_n)(k.s) : c \longrightarrow A_k(s) : c}$$

4. $$\frac{A(s) : c \sqcup \exists_X(d) \longrightarrow A'(s') : c'}{\exists_X^c A(s) : d \longrightarrow \exists_X^{c'} A'(s') : d \sqcup \exists_X(c')}$$

5. $P(X)(s) : c \longrightarrow A[X/Y](s) : c,$
   where Π contains $P(Y) :: A$
   and $A[X/Y] = \exists_\alpha(\alpha = X \wedge \exists_Y(\alpha = Y \wedge A))$.

Figure 8.1: Computation rules

8.4.1 Configurations and computation rules

A *configuration* is a triple $A(s) : c$ consisting of an agent A, an oracle s, and a finite constraint c (the store). The oracle will sometimes be omitted when it is either given by the context or not relevant. Given an agent A and an oracle s, we will sometimes refer to $A(s)$ as an *agent-oracle* pair. We define a binary relation $\longrightarrow$ over configurations according to Figure 8.1.

The behaviour of a selection is completely determined by the oracle. If the oracle begins with the number k, the selection can only take the kth branch. What happens if the test in the kth alternative never becomes true? This situation is addressed in the section on fairness.

The behaviour of a conjunction, for a given oracle, depends of course on the behaviour of the components. We use the functions π_k to split the oracle into oracles for each component.

8.4.2 Computations

The set of computations for a given program are defined as follows.

Definition 8.4.1 Assuming a program Π, a *computation* is an infinite sequence of configurations $(A_i(s_i) : c_i)_{i \in \omega}$ such that for all $i \ge 0$, we have either $A_i(s_i) : c_i \longrightarrow A_{i+1}(s_{i+1}) : c_{i+1}$ (*a computation step*), or $A_i(s_i) = A_{i+1}(s_{i+1})$, and $c_i \sqsubseteq c_{i+1}$ (*an input step*).

An input step from $A(s) : c$ to $A(s) : c'$, 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 $A_0(s_0) = A(s)$. □

8.4.3 Fairness

The definition of fairness for oracle-computations resembles the first definition of fairness, given in Section 4.5. The definition of inner computations is similar to Definition 4.5.1; the only difference is that we need to say what the oracle of an inner computation is.

Definition 8.4.2 Let the relation *immediate inner computation of* be the weakest relation over ω-sequences of configurations which satisfies the following.

1. $(A_i^k(\pi_k s_i) : c_i)_{i\in\omega}$ is an immediate inner computation of $(\bigwedge_{j\in I} A_i^j(s_i) : c_i)_{i\in\omega}$, 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 $(\exists_X^{c_i} A_i(s_i) : d_i)_{i\in\omega}$.

The relation '*inner computation of*' is defined to be the transitive closure over the relation 'immediate inner computation of'. □

Proposition 8.4.3 If $(A_i(s_i) : c_i)_{i\in\omega}$ is 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.

The definition of top-level fairness is also similar to the one given in section 4.5. The difference here is in the treatment of selections. An oracle-computation where the first agent is a selection is top-level fair if one of two things hold. Either the oracle begins with the number k, and the computation eventually selects the kth branch of the selection (this implicitly assumes that $1 \le k \le n$, where n is the number of alternatives of the selection), or the computation begins with a zero and none of the ask constraints ever become entailed by the store. The definition of initial fairness, and top-level fairness are the same as in the previous definition of fairness.

Definition 8.4.4 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 $i \ge 0$ such that $A_i \ne A_0$.

2. If $A_0 = c$, there is an $i \ge 0$ such that $c_i \sqsupseteq c$.

3. If $A_0 = (d_1 \Rightarrow B_1 [] \dots [] d_n \Rightarrow B_n)$, and $s = k.s'$ where $k \geq 1$, then there is an $i \geq 0$ such that $A_i = B_k$.

4. If $A_0 = (d_1 \Rightarrow B_1 [] \dots [] d_n \Rightarrow B_n)$, and $s = 0.s'$, then $d_j \not\sqsubseteq c_i$ 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. □

Lemmas 4.5.4 through 4.5.8 also apply for the oracle-based operational semantics.

8.5 Result and Trace Semantics

We extend the definitions of result and trace semantics to deal with oracles.

8.5.1 Results

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

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

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

8.5.2 Traces

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 $i \notin r$. 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 ith element of the store sequence of t.

The trace semantics of an agent A together with an oracle s, assuming a program Π, is defined as follows.

Definition 8.5.1

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

□

The trace semantics for computations with oracles satisfies the same compositional properties as the earlier computational model (see Section 4.7.3).

8.5.3 The abstract semantics

It is straight-forward to give an abstract semantics, completely analogous to the fully abstract semantics defined in a previous chapter.

Definition 8.5.2 For an agent A, an oracle s, and a program Π, let

$$\mathcal{A}_\Pi[\![A(s)]\!] = \{t \mid t \text{ is a subtrace of } t', \text{ for some } t' \in \mathcal{O}_\Pi[\![A(s)]\!]\}.$$

$\square$

8.6 An example

Consider the semantics of the agent

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

The result semantics is

$$\begin{array}{ll}
\mathcal{R}_\Pi[\![A(s)]\!]c = \{c\}, & \text{for oracles } s = 0\ldots \\
 & \;\text{and constraints } c \not\sqsupseteq (X=1),(Y=2) \\
\mathcal{R}_\Pi[\![A(s)]\!]c = \emptyset, & \text{for oracles } s = 0\ldots \\
 & \;\text{and constraints } c \sqsupseteq (X=1) \sqcup (Y=2) \\
\mathcal{R}_\Pi[\![A(s)]\!]c = \{c \sqcup (Z=3)\}, & \text{for oracles } s = 1\ldots \\
 & \;\text{and constraints } c \sqsupseteq (X=1) \\
\mathcal{R}_\Pi[\![A(s)]\!]c = \{c \sqcup (W=5)\}, & \text{for oracles } s = 2\ldots \\
 & \;\text{and constraints } c \sqsupseteq (Y=2) \\
\mathcal{R}_\Pi[\![A(s)]\!]c = \emptyset, & \text{in all other cases}
\end{array}$$

The idea behind the oracle semantics was to find a way to factor out non-determinism from ccp agents. We can see that for agent-oracle pairs $A(s)$ above, the result semantics is either an empty set or a singleton set, which supports the notion that agent-oracle pairs are deterministic.

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

$$\begin{array}{llllllll}
v(t) = (\bot, & \bot, & X=1, & X=1, & X=1, & (X=1)\sqcup(Z=3), & \ldots) \\
r(t) = \{ & & & 3, & 4 & & \}
\end{array}$$

The trace only has two computation steps. In the first step (step 0), nothing happens (an empty input step). In the following step (step 1) the constraint $X = 1$ is input, i.e, added to the store 'from the outside'. In the next step, nothing happens. In step number 3, the agent performs a computation step, 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. Even the trivial tell constraint $X = X$ has as operational semantics the uncountable set of traces $\{t \in \text{TRACE} \mid v(t)_i = v(t)_{i+1} \text{ whenever } i \in r(t)\}$.

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

$$\begin{aligned}
\mathcal{A}_\Pi[\![A(s)]\!] &= \{t \mid \mathrm{fn}\, t = \mathbf{id}, \lim t \notin \{X = 1\}^u \cup \{Y = 2\}^u\},\\
&\ \text{for oracles } s = 0 \ldots\\
\mathcal{A}_\Pi[\![A(s)]\!] &= \{t \mid \mathrm{fn}\, t \sqsubseteq (X = 1 \rightarrow Z = 3),\\
&\qquad \lim t \in \{X = 1\}^u \cap \{Z = 3\}^u\},\\
&\ \text{for oracles } s = 1 \ldots\\
\mathcal{A}_\Pi[\![A(s)]\!] &= \{t \mid \mathrm{fn}\, t \sqsubseteq (Y = 2 \rightarrow W = 5),\\
&\qquad \lim t \in \{Y = 2\}^u \cap \{W = 5\}^u\},\\
&\ \text{for oracles } s = 2 \ldots\\
\mathcal{A}_\Pi[\![A(s)]\!] &= \emptyset,\\
&\ \text{for oracles } s = k \ldots, \text{ with } k \geq 3
\end{aligned}$$

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

8.7 Confluence

When an agent-oracle pair is run, the oracle determines the non-deterministic choices in selections. The scheduling of agents in a conjunction is still a potential source of non-determinism. We would like to show that when we run an agent-oracle pair, the way we schedule the agents in conjunctions will not affect the final result. In other words, we want to show *confluence*.

In this section we consider a basic confluence property regarding finite computation sequences and a more general confluence property, which deals with (countably) infinite sets of arbitrary computations.

We begin by stating the generalised confluence theorem, which says that arbitrary countable sets of $A(s)$-computations may be combined. The rest of this chapter is devoted to the proof of the theorem.

Theorem 8.7.1 (Generalised confluence) Given an agent A, an oracle s and a constraint c such that, for $n \in \omega$, f_n is the functionality of some $A(s)$-computation with limit c. Let t be a trace such that $\lim t = c$ and $\mathrm{fn}\, t \sqsubseteq \bigsqcap_{n \in \omega} f_n$. Then there is a $A(s)$-computation with trace t', such that t is a subtrace of t'.

If there is also an initially fair $A(s)$-computation with limit c, there is an initially fair $A(s)$-computation with trace t', such that t is a subtrace of t'.

If there is a fair $A(s)$-computation with limit c, it follows that there is a fair $A(s)$-computation with trace t', such that t is a subtrace of t'.

The results of this chapter should be intuitively clear, and the reader may skip the rest of this chapter at first reading and turn directly to the chapter on fixpoint semantics.

8.7.1 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 complicates the proof of confluence. For example, consider the following configuration.

$$X = 5 \wedge \exists_Y^{\perp} Z = 7 : \perp$$

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

$$\begin{aligned} X = 5 \wedge \exists_Y^{\perp} Z = 7 : \perp &\longrightarrow X = 5 \wedge \exists_Y^{\perp} Z = 7 : X = 5 \\ &\longrightarrow X = 5 \wedge \exists_Y^{(X=5 \wedge Z=7)} Z = 7 : X = 5 \wedge Z = 7 \end{aligned}$$

If, on the other hand, we perform the reductions in the opposite order, we reach the following situation.

$$\begin{aligned} X = 5 \wedge \exists_Y^{\perp} Z = 7 : \perp &\longrightarrow X = 5 \wedge \exists_Y^{Z=7} Z = 7 : Z = 7 \\ &\longrightarrow X = 5 \wedge \exists_Y^{Z=7} Z = 7 : X = 5 \wedge Z = 7 \end{aligned}$$

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 $\lceil \cdot \rceil_c$ 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 $\lceil A \rceil_c$, the constraint c represents the information which is available globally. The mapping will be used to define an equivalence relation over agents.

Definition 8.7.2 For an agent A and a constraint c, let $\lceil A \rceil_c$ be the agent given by the following rules.

1. $\lceil \exists_X^d A \rceil_c = \exists_X^e \lceil A \rceil_e$, where $e = d \sqcup \exists_X c$.
2. $\left\lceil \bigwedge_{j \in I} A^j \right\rceil_c = \bigwedge_{j \in I} \lceil A^j \rceil_c$.
3. $\lceil A \rceil_c = A$, if A is not an existential quantification or a conjunction.

For configurations $A : c$ and $B : d$, say that $A : c \equiv B : d$, if $c = d$ and $\lceil A \rceil_c = \lceil B \rceil_d$. □

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

It is easy to establish that if $A : c \longrightarrow B : d$ and $A : c \equiv A' : c'$ then there is some configuration $B' : d'$ such that $A' : c' \longrightarrow B' : d'$ 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, $\lceil A \rceil_c : c$, represent the abstract configuration (equivalence class) containing $A : c$. For an abstract configuration $K = [A(s) : c]$, let $\mathit{store}(K) = c$, and $\mathit{input}(K, d) = [A(s) : c \sqcup d]$. For abstract configurations K and L say that $K \Rightarrow L$ if there are $(A(s) : c) \in K$ and $A'(s') : c' \in L$ such that $A(s) : c \longrightarrow^* A'(s') : c'$. Say that $K \rightsquigarrow L$ if $\mathit{input}(K, c) \Rightarrow L$, for some constraint c.

8.7.2 Finite confluence

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

Proposition 8.7.3 Suppose c is a constraint and K and L are abstract configurations. If $K \Rightarrow L$ we have $\mathit{input}(K, c) \Rightarrow \mathit{input}(L, c)$. If $K \Rightarrow L$ in one step we have $\mathit{input}(K, c) \Rightarrow \mathit{input}(L, c)$ 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 $\rightsquigarrow$-relation is transitive, as expected.

Lemma 8.7.4 (Finite 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 only consider the case when $K \Rightarrow L$ and $K \Rightarrow M$ in one computation step; it is easy to show the general case using induction on the length of the computation sequences. The proof is by induction on the agent of K. The details are given in Section 8.9. □

The confluence theorem can be seen as asserting the existence of a partial binary function $*$ over abstract configurations. The function is defined for abstract configurations L and M such that $K \Rightarrow L, M$, for some K, and satisfies

$$L \Rightarrow L * M \quad \text{and} \quad M \Rightarrow L * M.$$

Clearly, $\mathit{store}(L * M) \sqsupseteq \mathit{store}(L), \mathit{store}(M)$.

8.7.3 Chains

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 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*.

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

Also, 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, we say that a chain is *initially fair*, if one can form an initially fair computation from the chain.

8.7.4 Properties of chains

First we consider a simple property of chains.

Proposition 8.7.5 Let $(K_i)_{i \in \omega}$ be a chain and c a constraint such that $c \sqsubseteq \mathit{store}(K_n)$, for some n. Let $L = \mathit{input}(K_0, c)$. It follows that $L \rightsquigarrow K_n$.

Proof. Clearly,

$$L = \mathit{input}(K_0, c) \rightsquigarrow \mathit{input}(K_1, c) \rightsquigarrow \ldots \rightsquigarrow \mathit{input}(K_n, c) = K_n.$$

□

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 8.7.6 A $\rightsquigarrow$-chain $(K_i)_{i \in \omega}$ is *maximal* if for any abstract configuration L such that $K_i \Rightarrow L$, for some i, there is an $n \in \omega$ such that $L \rightsquigarrow K_n$. □

Lemma 8.7.7 Any fair chain is maximal.

Proof. Consider a fair chain $(K_i)_{i\in\omega}$. Since any suffix of a fair chain is fair, we only need to show that when $K_0 \Rightarrow L$, there is some $n \in \omega$ such that $L \rightsquigarrow K_n$. The proof is by induction on the agent of K_0, and is given in Section 8.9. □

The converse of Lemma 8.7.7 does not hold; there are maximal chains which are not fair. For example, let K be the configuration

$$(\mathbf{true} \Rightarrow A)(0.s) : c,$$

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

Corollary 8.7.8 Let $(K_i)_{i\in\omega}$ be a fair chain. Let L be such that $K_0 \rightsquigarrow L$ and $\mathit{store}(L) \sqsubseteq \lim(K_i)_{i\in\omega}$. There is an $n \in \omega$ such that $L \rightsquigarrow K_n$.

Proof. Let $c = \mathit{store}(L)$. For $i \in \omega$, let $K_i' = \mathit{input}(K_i, c)$. Clearly, $(K_i')_{i\in\omega}$ is a fair chain. We have $K_0' \Rightarrow L$. Since $(K_i')_{i\in\omega}$ is maximal we have $L \rightsquigarrow K_n'$, for some n. Choose m such that $m \geq n$ and $\mathit{store}(K_m) \sqsupseteq c$. By transitivity we have $L \rightsquigarrow K_m'$. Since $K_m' = K_m$ we have $L \rightsquigarrow K_m$. □

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 8.7.9 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 $K_i \rightsquigarrow L_j$. 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, which is by induction on the agent of K_0, is given in Section 8.9. □

Lemma 8.7.10 Suppose that we have chains $(K_i)_{i\in\omega}$ and $(L_i)_{i\in\omega}$ such that $K_0 = L_0$, $\lim(K_i)_{i\in\omega} = \lim(L_i)_{i\in\omega}$ and for all $i \in \omega$ there is a $j \in \omega$ such that $K_i \rightsquigarrow L_j$. 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 $K_0 \rightsquigarrow L_i$. Thus, $L_i \rightsquigarrow K_n$, for some n. The chain

$$L_i \rightsquigarrow K_n \rightsquigarrow K_{n+1} \rightsquigarrow \ldots$$

is fair, since it has a fair suffix. Proposition 8.7.9 is applicable, since obviously $L_i \rightsquigarrow L_j$, for some j, and by assumption, $K_n \rightsquigarrow L_j$, for some j and so on. It follows that the chain

$$L_i \rightsquigarrow L_{i+1} \rightsquigarrow L_{i+2} \rightsquigarrow \ldots$$

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

8.7.5 Construction of an input-free computation

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

Lemma 8.7.11 Given an agent A, an oracle s and a constraint c such that, for $n \in \omega$, f_n is the functionality of some $A(s)$-computation with limit c. Suppose $(\bot \to c) \sqsubseteq \bigcap_{n\in\omega} f_n$. It follows that there is an $A(s)$-computation with functionality $(\bot \to c)$ 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 $(\bot \to c)$ and limit c.

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

Proof. Given $A(s)$ and c as in the lemma above. For all n, we can construct a chain $\{K_i^n\}_{i\in\omega}$ which corresponds the computation which has f_n as functionality. We assume that $K_0^n = A(s) : \bot$, for all n (since for any given computation we can form a similar computation where the initial environment is equal to $\bot$), and that for all n and i, either $K_i^n \Rightarrow K_{i+1}^n$ or $\mathit{input}(K_i^n, d) = K_{i+1}^n$, 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 $(\bot \to c)$ and limit c. For each n and i we will have $K_i^n \Rightarrow L_j$, for some j. We will use the notation $K \overset{d}{\rightsquigarrow} L$ if $\mathit{input}(K, d) \Rightarrow \mathit{input}(L, d)$.

Let p be a function $p : \omega \to \omega$ 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 $c_i = \mathit{store}(L_i)$.

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

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

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

There is alway at least one m which satisfies the above, since we always have $K_0^{p(i)} \rightsquigarrow L_i$, furthermore there always a maximal m since for some m we have $\mathit{store}(K_m^{p(i)}) \sqsupseteq c_i$.

Let $d = \sqcup_{i\in\omega} \mathit{store}(L_i)$. We would like d to be equal to c. Suppose it is not. It follows that $d \notin f_n$, for some n. Let j be the greatest such that $K_j^n \Rightarrow L_i$, for some i. If $K_j^n \Rightarrow K_{j+1}^n$, we know from the way the L_is were selected that $K_{j+1}^n \Rightarrow L_{j'}$, for some j', which contradicts the assumption about j. So the step from K_j^n to K_{j+1}^n is an input step. If $\mathit{store}(K_{j+1}^n) \sqsubseteq d$, we would expect $K_j^n \overset{c_i}{\rightsquigarrow} K_j^n$, for some i, which again leads to a contradiction. So we conclude that $\mathit{store}(K_{j+1}^n) \not\sqsubseteq d$, but then d must be a fixpoint of f_n, and we have arrived at a contradiction. We can conclude that $\sqcup_{i\in\omega} \mathit{store}(L_i) = c$.

To establish fairness properties, note that for each i there is a j such that $K_i^0 \rightsquigarrow L_j$. If $(K_i^0)_{i\in\omega}$ is initially fair, it follows by Proposition 8.7.9 that $(L_i)_{i\in\omega}$ is initially fair. Similarly, if $(K_i^0)_{i\in\omega}$ is fair, it follows by Lemma 8.7.10 that $(L_i)_{i\in\omega}$ is fair. □

8.7.6 Proof of the general confluence theorem

In the proof below, we will use the following notation. Given a trace t write $[t]$ for the agent

$$\bigwedge_{i\notin r(t)} (v(t)_i \Rightarrow v(t)_{i+1}).$$

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

Proof. [of Theorem 8.7.1] 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 $A' = A \wedge [t]$. 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, 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 $\bigcap_{n\in\omega} f'_n \sqsupseteq (\bot \to c)$.

First, let $f'_{n+1} = f_n$, for $n \in \omega$.

Second, note that the agent $[t]$ has a computation with functionality g which is essentially the inverse of $\mathrm{fn}\, t$. 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 $[t]$ 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 $\bigcap_{n\in\omega} f'_n \sqsupseteq \bigcap_{n>0} f'_n = g \cap (\bigcap_{n\in\omega} f_n) \sqsupseteq (\bot \to c)$. It follows by Lemma 8.7.11 that there is a $A'(s')$-computation with functionality $(\bot \to c)$ and limit c.

In other words, there is a trace $u \in \mathcal{O}_\Pi[\![A \wedge [\bar{t}]]\!]$ such that $\lim u = c$ and $\mathrm{fn}\, u = (\bot \to c)$. It follows that $u = u_1 \vee u_2$, where $u_1 \in \mathcal{O}_\Pi[\![A]\!]$ and $u_2 \in \mathcal{O}_\Pi[\![[\bar{t}]]\!]$. Suppose $d \sqsubseteq c$ such that $d \in \mathrm{fn}\, u_1$. It follows that $d \notin \mathrm{fn}\, u_2$, by the computation rules. Since $\mathrm{fn}\, u_2 \sqsubseteq \mathrm{fn}\, \bar{t}$, we have $d \notin \mathrm{fn}\, \bar{t}$. By the definition of $\bar{t}$ it follows that $d \in t$. So $\mathrm{fn}\, u_1 \subseteq \mathrm{fn}\, t$, i.e., $\mathrm{fn}\, u_1 \sqsupseteq \mathrm{fn}\, t$. It follows that t is a subtrace of u_1 and we are done. □

8.8 Concluding Remarks

The idea behind the oracle semantics is the notion that the non-determinism in concurrent constraint programming stems from the non-deterministic selection. By introducing oracles to control the behaviour of the non-deterministic choice, we can effectively isolate the non-deterministic component and thus view the behaviour of an agent as a set of *deterministic* behaviours, indexed by the oracles.

The fact that the behaviour of an agent is deterministic, for a given oracle, implies that there should be some kind of confluence property for agent-oracle pairs. We have seen in this chapter both a simple, finite confluence property, and a generalised form of confluence.

The finite confluence property is analogous to the Church-Rosser property of lambda-calculus. But it is not sufficient to only consider finite computations. The generalised confluence property (Theorem 8.7.1) shows how sets of computations can be combined into stronger computations. An important aspect of the generalised confluence theorem is that the combination of computations preserves fairness. Also, the generalised confluence theorem is not restricted to finite sets of computations, but can also be applied to (countably) infinite sets of computations.

The confluence properties will be essential in the correctness proof of the fixpoint semantics presented in the next chapter.

8.9 Proofs from Chapter 8

Proof of Lemma 8.7.4 (finite confluence)

We will only consider the case when $K \Rightarrow L$ and $K \Rightarrow M$ in one computation step, and show that in this case there is a configuration that can be reached from L and M in one 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, 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 performed by 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

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

and

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

(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, A^l, and one agent consisting of all other components in the conjunction. We also re-order the oracle in the same manner. It follows directly from the operational semantics that this re-ordering does not affect the operational behaviour of the agent. Let $A^* = \bigwedge_{j \in I'} A^j$, where $I' = I \setminus \{k, l\}$. We assume that A can be written on the form $A^k \wedge A^l \wedge A^*$. We have

$$A^k \wedge A^l \wedge A^*(s) : c \longrightarrow B^k \wedge A^l \wedge A^*(s') : c'$$

and

$$A^k \wedge A^l \wedge A^*(s) : c \longrightarrow A^k \wedge B^l \wedge A^*(s'') : c''.$$

By Proposition 8.7.3 it follows that

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

and

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

Let s''' be such that $\pi_* s''' = \pi_* s$, $\pi_k s''' = \pi_k s'$, and $\pi_l s''' = \pi_l s''$. Also, let $c''' = c' \sqcup c''$. We have

$$B^k \wedge A^l \wedge A^*(s') : c' \longrightarrow B^k \wedge B^l \wedge A^*(s''') : c'''$$

and

$$A^k \wedge B^l \wedge A^*(s'') : c'' \longrightarrow A^k \wedge A^l \wedge A^*(s''') : c'''.$$

Suppose K is of the form $\exists_X^{c_0'} A_0' : c_0$. It follows that $L = \exists_X^{c'} A' : c$ and $M = \exists_X^{d'} B' : d$. We can assume that $c' \sqsupseteq \exists_X(c)$ and $d' \sqsupseteq \exists_X(d)$.

By the induction hypothesis it follows that there is an agent C, an oracle s'' and a constraint e such that $A'(s) : c' \longrightarrow C(s'') : e$ and $B'(s') : d' \longrightarrow C(s'') : e$. By the computation rules it follows that

$$\exists_X^{c'} A'(s) : c \longrightarrow \exists_X^{e} C(s'') : c \sqcup \exists_X(e)$$

and

$$\exists_X^{d'} B'(s'') : d \longrightarrow \exists_X^{e} C(s'') : d \sqcup \exists_X(e).$$

To show that $c \sqcup \exists_X(e) = d \sqcup \exists_X(e)$, recall that we assumed that $K \Rightarrow L$ and $K \Rightarrow M$ in one computation step. Let $c_0 = \mathit{store}(K)$. By the computation rules we have $c = c_0 \sqcup \exists_X(c')$ and $d = c_0 \sqcup \exists_X(d')$. Since $c' \sqsubseteq e$ it follows that $\exists_X(c') \sqsubseteq \exists_X(e)$ and thus $c \sqcup \exists_X(e) = c_0 \sqcup \exists_X(c') \sqcup \exists_X(e) = c_0 \sqcup \exists_X(e)$. In a similar way we can establish that $d \sqcup \exists_X(e) = c_0 \sqcup \exists_X(e)$.

Proof of Lemma 8.7.7

In the proofs below the following notation is used. We write $\text{FACTOR}_k(K) = L$ when $K = [\bigwedge_{j \in I} A^j(s) : c]$, $k \in I$, and $L = [A^k(\pi_k s) : c]$. In the same way we write $\text{LOCAL}(K) = L$ when $K = [\exists_X^c A(s) : d]$ and $L = [A(s) : c \sqcup \exists_X(d)]$. It is straight-forward to establish that FACTOR and LOCAL are well-defined.

Let $(K_i)_{i \in \omega}$ be a fair chain. We will only consider the case when $K_0 \Rightarrow L$, 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, i.e, $K_0 = [c(s) : d]$. 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 $(K_i)_{i \in \omega}$, given by $\text{FACTOR}_k(K_i)_{i \in \omega} = (K_i')_{i \in \omega}$. This computation is fair, since it is an inner computation of a fair computation. By the computation rules, L must be of the form $\bigwedge_{j \in I} A^j(s) : c$. Let $L' = [A^k(\pi_k s) : c]$. Because of the computation rules, we have either $K_0' \Rightarrow L'$, or that the step from K_0' to L' is an input step. If $K_0' \Rightarrow L'$, we can apply the induction hypothesis and conclude that $L' \rightsquigarrow K_n'$, for some n. Similarly, if the step from K_0' to L' is an input step we apply the induction hypothesis and Proposition 8.7.5 and find that $L' \rightsquigarrow K_n'$.

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

If K_0 is a selection $(c_1 \Rightarrow A_1 \,[]\, \dots \,[]\, c_n \Rightarrow A_n)$ alternatives, and the first element of the oracle of K_0 is k, 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 $L \rightsquigarrow K_n$.

Suppose the agent of K_0 is an existential quantification. Let the chain $(K'_i)_{i\in\omega} = \text{LOCAL}(K_i)_{i\in\omega}$, and if $L = [\exists^c_X A(s) : d]$ 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 $L' \Rightarrow K'_n$, 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 $L \rightsquigarrow K_n$ through an input step.

Proof of Proposition 8.7.9

The proof is by induction of the agent of L_0, 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 $\mathit{store}(K_i) \sqsupseteq c$. Since $K_i \rightsquigarrow L_j$, for some j, we have $\mathit{store}(K_i) \sqsubseteq \mathit{store}(L_j)$, thus $c \sqsubseteq \mathit{store}(L_j)$ and $(L_i)_{i\in\omega}$ is initially fair.

Suppose the agent of L_0 is a conjunction. Consider an inner computation of $(L_i)_{i\in\omega}$, given by $\text{FACTOR}_k(L_i)_{i\in\omega} = (L'_i)_{i\in\omega}$. If we let $(K'_i)_{i\in\omega} = \text{FACTOR}_k(K_i)_{i\in\omega}$, 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 $K_i \rightsquigarrow L_j$, for some j. By the computation rules, we have $K'_i \rightsquigarrow L'_j$. 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, where $1 \leq k \leq n$. Since $K_0 = L_0$, and because of fairness there must be an i such that the agent of K_i is the agent of the kth alternative of the selection. Since $K_i \rightsquigarrow L_j$, 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, and since the only computation step that a selection can perform is to chose 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, and the oracle of L_0 begins with k, 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 $(L_i')_{i\in\omega} = \textsc{local}(L_i)_{i\in\omega}$, and, in the same way, $(K_i')_{i\in\omega} = \textsc{local}(K_i)_{i\in\omega}$. 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 $K_i \rightsquigarrow L_j$, for some j. Thus, $K_i = [\exists_X^c A(s) : d]$ and $L_j = [\exists_X^{c'} A'(s') : d']$, for a variable X, and appropriately selected agents, constraints and oracles, and we know that

$$\exists_X^c A(s) : d \sqcup e \longrightarrow^* \exists_X^{c'} A'(s') : d',$$

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

$$A(s) : c \sqcup \exists_X(d \sqcup e) \longrightarrow A'(s') : c',$$

and $d' = d \sqcup e \sqcup \exists_X(c')$. We have $c' \sqsupseteq \exists_X(d \sqcup e)$, and thus $c' \sqsupseteq \exists_X(d \sqcup e) \sqcup \exists_X(c') = \exists_X(d \sqcup e \sqcup \exists_X(c')) = \exists_X(d')$. It follows that $c' = c' \sqcup \exists_X(d')$. Since $K_i' = A(s) : c \sqcup \exists_X(d)$ and $L_j' = A'(s') : c' \sqcup \exists_X(d')$ we have $K_i' \rightsquigarrow L_j'$. Since $K_0' = L_0'$ we can apply the induction hypothesis and conclude that $(L_i')_{i\in\omega}$ is initially fair. Since the inner computation of $(L_i)_{i\in\omega}$ it follows that $(L_i)_{i\in\omega}$ is initially fair.

Suppose the agent of L_0 is a call. There is some i such that K_i is the body of the corresponding procedure, wrapped in existential quantifications to model parameter passing. There is a $j \in \omega$ such that $K_i \rightsquigarrow L_j$. The agent of L_j cannot be a call, because of the computation rules. It follows that $(L_i)_{i\in\omega}$ contains at least one computation step, and since the only computation step a call can perform is the reduction to the body of the corresponding procedure, it follows that $(L_i)_{i\in\omega}$ is initially fair.

Chapter 9

Fixpoint Semantics

In this chapter, we consider the problem of giving a fixpoint semantics for concurrent constraint programming. As shown in Chapter 7 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 so the best one can hope for is a fixpoint semantics together with a simple abstraction operator.

The fixpoint semantics is based on the oracle semantics presented in the previous chapter.

9.1 Introduction

For an agent-oracle pair $A(s)$ 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 an agent-oracle pair $A(s)$, 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, 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-oracle pair $A(s)$ 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 $A(s)$. We will also find that this information is sufficient to give a compositional semantics.

It would seem that a domain where the elements are pairs consisting of a window and a functionality would be a promising candidate for a fixpoint semantics. However, as we shall see in the following section it turns out that it is not possible to find an ordering in which the existential quantifier is monotone (or continuous).

9.2 There is no fully abstract fixpoint semantics for agent-oracle pairs

In this section we will examine the problems of giving a fully abstract fixpoint semantics for agent-oracle pairs $A(s)$. Keep in mind that a fully abstract fixpoint of agent-oracle pairs must satisfy the following requirements.

1. Like in any other fixpoint semantics, the semantics of a procedure P with the definition $P :: P$ must be the least element of the semantic domain.

2. Since the behaviour of the agent **true** is the same as the behaviour of P, as defined above, it follows that the semantics of **true** is also the least element of the domain.

3. As in any other fixpoint semantics, we expect all operations to be monotone.

4. The semantic domain should capture both the *functionality*, which describes the output of the agent, and the *window*, which describes the set of requirements the agent imposes on the input.

As stated above, the semantics of $\mathbf{true}(s)$ (regardless of the choice of the oracle s) must be the least element of the domain.

Now, let the agent A be

$$\mathbf{true} \wedge (X = 3 \Rightarrow \mathbf{true} \; [] \ldots)$$

and the agent B

$$X = 3 \wedge (X = 3 \Rightarrow \mathbf{true} \; [] \ldots).$$

(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 $\pi_2 s = 1.s'$, for some oracle s'. 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 **true** is weaker than $X = 3$, but when we look at $\exists_X B(s)$ we see that the semantics of $\exists_X B(s)$ is equal to **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 it, when quantified with X, still implies that $X = 3$. In other words, we must have $\exists_X c \sqsupseteq (X = 3)$. It is easy to see that the only c for which this holds is $c = \top$. In other words, he 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, 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.

Proposition 9.2.1 There is no fully abstract fixpoint semantics for agent-oracle pairs.

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 fixpoint semantics [2, 4] since they considered non-deterministic programming languages.

9.3 Hiding

Because of the discovery that it is not possible to give a fully abstract semantics for agent-oracle pairs, 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 conjunction we introduce different kinds of renamings.

The following section presents the appropriate types of renamings and gives some of their properties. The proofs are given in Section 9.11.

9.3.1 Renamings

Recall that the set of formulas of a pre-constraint system is 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 9.3.1 A *renaming* θ is a mapping $\theta : \mathit{Var} \to \mathit{Var}$ over variables. An *injective renaming* is a renaming which is an injective function over variables. □

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

We assume that a truth assignment $\models$ satisfies the following, for an assignment V, a renaming θ and a formula ϕ.

$$V \models \theta\phi \text{ iff } V \circ \theta \models \phi$$

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. $\theta[\phi] = [\theta\phi]$, for formulas ϕ.
2. $\theta(\bigsqcup R) = \bigsqcup_{d \in R} \theta d$, for directed sets $R \subseteq \mathcal{K}(\mathcal{U})$.

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 θ we have $V \models \theta c$ iff $V \circ \theta \models c$.

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

Proposition 9.3.2 An injective renaming θ is also injective when seen as a function over constraints.

For a renaming θ let θ^{-1} be the upper adjoint of θ, that is, let θ^{-1} be the monotone function over constraints such that $c \sqsubseteq \theta^{-1}(\theta c)$ and $\theta(\theta^{-1}c) \sqsubseteq c$.

Since each renaming distributes over $\bigsqcup$ it follows that θ^{-1} is well-defined and can be given explicitly by

$$\theta^{-1}c = \bigsqcup\{d \mid \theta d \sqsubseteq c\}.$$

We also have $c \sqsubseteq \theta^{-1}d$ iff $\theta c \sqsubseteq d$, for constraints c, d.

For an injective renaming θ we have $\theta^{-1} \circ \theta = \mathbf{id}$.

9.3.2 Hidden variabes

In the following text, we assume that the variables are split into two sets, the *visible* variables and the set H of *hidden* variables. We assume that hidden variables do not occur in agents, programs, traces or computations. The hidden variables will be used in the fixpoint semantics 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 agents, programs, traces and computations.)

We need three types of operations on hidden variables. The first operation is $\exists_H$. Let $\exists_H c$ be the existential quantification of all hidden variables occurring in the constraint c.

The second operation is the renaming new_X, for a visible variable X. We will use new_X to model the existential quantification of variables.

Let new_X be an injective renaming which

1. maps X to a hidden variable, and
2. maps every visible variable distinct from X to itself, and
3. maps every hidden variable to a hidden variable.

Since new_X is assumed to be injective there is an inverse new_X^{-1} which satisfies $\mathsf{new}_X^{-1} \circ \mathsf{new}_X = \mathbf{id}$ and $\mathsf{new}_X \circ \mathsf{new}_X^{-1} \sqsubseteq \mathbf{id}$.

The third renaming is used when giving the semantics of a 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 renamings (called *projections*) $\{\theta_j\}_{j \in I}$ such that (writing $\theta_j H$ for $\{\theta_j X \mid X \in H\}$)

1. $(\theta_j H) \bigcap (\theta_{j'} H) = \emptyset$, for $j \neq j'$, and
2. $\theta_j X = X$, for visible variables X.

Proposition 9.3.3 Let X be a visible variable, c a constraint independent of hidden variables, and j, k be members of some set I such that $j \neq k$. It follows that

1. $\exists_H \circ \mathsf{new}_X = \exists_H \circ \exists_X$,
2. $\mathsf{new}_X(\exists_X c) \sqsubseteq c$,
3. $\mathsf{new}_X^{-1} c = \exists_X c$,
4. $\theta_j c = \theta_j^{-1} c = c$, and

5. $\theta_j^{-1} \circ \theta_k = \exists_H$.

(The proofs are given in Section 9.11.)

9.3.3 Applying renamings on sets and closure operators

Injective renamings can be generalized to sets and closure operators.

For an injective renaming θ, and a set of constraints S, let $\theta S = \{c \mid \theta^{-1} c \in S\}$.

Proposition 9.3.4 For a closure operator f, and an injective renaming θ, we have $c \in \theta f$ iff c is a fixpoint of $\theta \circ f \circ \theta^{-1} \sqcup \mathbf{id}$.

Proposition 9.3.5 Let S be a set of constraints and X a variable. It follows that $\mathrm{E}_H(\mathrm{E}_X S) = \mathrm{E}_H(\mathsf{new}_X S)$.

The following proposition formalises the idea that applying projections $\{\theta_j\}_{j \in I}$ to a family of closure operators $\{f_j\}_{j \in I}$ will guarantee that they are independent with respect to their hidden variables.

Proposition 9.3.6 Let $\{f_j\}_{j \in I}$ be a family of closure operators. Let $f = \bigcap_{j \in I} \theta_j f_j$, and c be such that $\exists_H(fc) = c$. Let $d = \bigsqcup_{j \in I}(\theta_j f_j)c$. It follows that $d = fc$.

9.4 Trace bundles

Here, we define the domain of the fixpoint semantics.

Let CL with typical element f be the lattice of closure operators over $\mathcal{U}$, and let W with typical element w be the lattice of windows over $\mathcal{U}$, i.e., with elements $\wp(\mathcal{U})$ ordered by reverse inclusion. Let BUNDLE, the *trace bundles*, be the set of pairs $\langle f, w \rangle$ in CL $\times$ W.

For a trace bundle $\langle f, w \rangle$, let $\mathsf{F}\langle f, w \rangle = f$, and $\mathsf{W}\langle f, w \rangle = w$. Let $\sqsubseteq \subseteq$ BUNDLE $\times$ BUNDLE be defined so that $\langle f, w \rangle \sqsubseteq \langle f', w', \rangle$ iff $f \sqsubseteq f'$, and $w \supseteq w'$.

Under this ordering, BUNDLE forms a complete lattice, with $\bot = \langle \mathbf{id}, \mathcal{U} \rangle$, $\top = \langle \bot \to \top, \emptyset \rangle$, and

$$\langle f_1, w_1 \rangle \sqcup \langle f_2, w_2 \rangle = \langle f_1 \cap f_2, w_1 \cap w_2 \rangle .$$

The semantics of an agent can now be given as a continuous function from oracles to trace bundles. Let $\mathsf{A} = (\text{ORACLE} \to \text{BUNDLE})$.

9.5 The Least-fixpoint Semantics

We are now ready for the first fixpoint semantics. We begin by defining a set of basic operations, corresponding to the program constructs of ccp.

9.5.1 Basic Operations

Tell constraints First, to give the semantics of a tell constraint c we define $(\!|c|\!)$ 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

$$(\!|c|\!) = \langle \bot \to c, \{c\}^u \rangle \, .$$

Parallel Composition Given a family of trace bundles $\{\langle f_j, w_j \rangle\}_{j \in I}$, 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.

$$\left\langle \bigcap_{j \in I} \theta_j f_j, \bigcap_{j \in I} \theta_j w_j \right\rangle$$

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 : $\mathcal{U} \to$ BUNDLE $\to$ 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

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

We lift the definition of select from BUNDLE to A by

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

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

For the case when in a selection no alternative is ever chosen, we define a function unless : $\mathcal{U}^n \to$ BUNDLE, for $n \geq 0$, as follows. Given constraints $c_1, \ldots, c_n$ let

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

Definition of $\mathsf{E}[\![A]\!] : \mathsf{A}^{\mathcal{N}} \to \mathsf{A}$

$$\begin{aligned}
\mathsf{E}[\![c]\!]\sigma s &= \llparenthesis c \rrparenthesis \\
\mathsf{E}[\![\textstyle\bigwedge_{j\in I} A^j]\!]\sigma s &= \left\langle \bigcap_{j\in I} \theta_j f_j, \bigcap_{j\in I} \theta_j w_j \right\rangle ,
\end{aligned}$$

where $\mathsf{E}[\![A^j]\!]\sigma(\pi_j s) = \langle f_j, w_j\rangle$, for $j \in I$

$$\begin{aligned}
\mathsf{E}[\![\square_{k\leq n}\, c_k \Rightarrow A_k]\!]\sigma s &= \begin{cases} \text{unless}(c_1,\ldots,c_n), & \text{if } s = 0s' \\ \text{select}\, c_k(\mathsf{E}[\![A_k]\!]\sigma)s', & \text{if } s = ks', \text{ for } 1 \leq k \leq n \\ \langle \mathbf{id}, \emptyset\rangle\, , & \text{otherwise} \end{cases} \\
\mathsf{E}[\![\exists_X A]\!]\sigma s &= \mathrm{E}_X(\mathsf{E}[\![A]\!]\sigma s) \\
\mathsf{E}[\![p(X)]\!]\sigma s &= \{\alpha \to X\}(\sigma p s)
\end{aligned}$$

Definition of $\mathsf{P}[\![\Pi]\!] : \mathsf{A}^{\mathcal{N}} \to \mathsf{A}^{\mathcal{N}}$

$$\mathsf{P}[\![\Pi]\!]\sigma P s = \{Y \to \alpha\}(\mathsf{E}[\![A]\!]\sigma s),$$

where for each $P \in \mathcal{N}$ the definition in Π is assumed to be of the form $P(Y) :: A$, for some variable Y and some agent A

Figure 9.1: The oracle fixpoint semantics

Existential quantification the function E_X : BUNDLE $\to$ BUNDLE which gives the trace bundle for $\exists_X A(s)$, given the trace bundle for $A(s)$.

Let

$$\mathrm{E}_X\,\langle f, w\rangle = \langle \mathsf{new}_X f, \mathsf{new}_X w, \rangle\,.$$

The following proposition expresses a simple relationship between the functionality and window of an agent, for a given oracle.

Proposition 9.5.1 Given σ such that σ is the least fixpoint of $\mathsf{P}[\![\Pi]\!]$, for some program Π, it holds that $w \subseteq f$, where $\langle f, w\rangle = \mathsf{E}[\![A]\!]\sigma s$.

It is straight-forward to verify the proposition by examination of the semantic functions.

9.5.2 Fixpoint Semantics

The oracle fixpoint semantics is given in Figure 9.1.

9.6 Examples

We give the fixpoint semantics of some programs.

Example 9.6.1 (erratic) Suppose the program Π contains the definition of the erratic procedure (Section 3.9), and that σ is the least fixpoint of $\mathsf{P}[\![\Pi]\!]$. We now have the following cases.

1. $\mathsf{E}[\![\mathrm{erratic}(X)]\!]\sigma s = \langle(\bot \rightarrow X = 0), \{X = 0\}^u\rangle$, if $s = 1.s'$, for some s'.
2. $\mathsf{E}[\![\mathrm{erratic}(X)]\!]\sigma s = \langle(\bot \rightarrow X = 1), \{X = 1\}^u\rangle$, if $s = 2.s'$, for some s'.
3. $\mathsf{E}[\![\mathrm{erratic}(X)]\!]\sigma s = \langle \mathbf{id}, \emptyset\rangle$, otherwise.

Note that only in the cases when s begins with a 1 or a 2 is the window non-empty. This indicates that the erratic procedure will always select either of the two first alternatives. In these alternatives, applying the functionality to any constraint gives a result which lies in the window. This indicates that no conditions are imposed on the input. □

Example 9.6.2 (merge) Assume that the program Π contains the definition of procedure merge (Section 3.11) and that σ is equal to $\mathsf{P}[\![\Pi]\!]\sigma'$, for some σ'.

Now, for variables X, Y, and Z, and oracle s, the semantics of the call merge(X, Y, Z) is as follows.

1. If $s = 1.s'$, we have $\mathsf{E}[\![\mathrm{merge}(X, Y, Z)]\!]\sigma s = \langle f_1, w_1\rangle$, where

$$\begin{aligned} f_1 = (&\exists_A \exists_{X_1}(X = [A \mid X_1]) \\ &\rightarrow \mathsf{new}_A \mathsf{new}_{X_1} \mathsf{new}_{Z_1}((\bot \rightarrow X = [A \mid X_1]) \\ &\qquad\qquad\qquad\qquad \cap (\bot \rightarrow Z = [A \mid Z_1]) \\ &\qquad\qquad\qquad\qquad \cap f_r)), \end{aligned}$$

and

$$\begin{aligned} w_1 = \mathsf{new}_A \mathsf{new}_{X_1} \mathsf{new}_{Z_1}(&\{X = [A \mid X_1]\}^u \\ &\cap \{Z = [A \mid Z_1]\}^u \\ &\cap w_r), \end{aligned}$$

and $\langle f_r, w_r\rangle = \{X, Y, Z \rightarrow X_1, Y, Z_1\}(\sigma' \text{ merge } s')$.

2. If $s = 3.s'$, we have $\mathsf{E}[\![\mathrm{merge}(X, Y, Z)]\!]\sigma s = \langle f_3, w_3\rangle$, where

$$f_3 = (X = [\,] \rightarrow Z = Y)$$

and

$$w_3 = \{X = [\,]\}^u \cap \{Z = Y\}^u.$$

3. If $s = 0.s'$, we have $\mathsf{E}[\![\mathrm{merge}(X,Y,Z)]\!]\sigma s = \langle f_0, w_0 \rangle$, where

$$f_0 = \mathbf{id}$$

and

$$\begin{aligned} w_0 = \mathcal{U} &\setminus \{\exists_A \exists_{X_1}(X = [A \mid X_1])\}^u \\ &\setminus \{\exists_A \exists_{Y_1}(Y = [A \mid Y_1])\}^u \\ &\setminus \{X = [\,]\}^u \\ &\setminus \{Y = [\,]\}^u \,\rangle\,. \end{aligned}$$

The case when $s = 2.s'$ is omitted, since it and the case $s = 1.s'$ are symmetric. Similarily, the cases $s = 4.s'$ and $s = 3.s'$ are symmetric.

Item 3 reflects the case when the call remains passive. As expressed in the window, this may happen when neither X nor Y become bound to lists.

Item 2 describes the case when X becomes bound to an empty list. The functionality says that the agent may bind Z to Y, when X has become bound to an empty list, and the window says that X must become bound to a list, and Z must become bound to Y.

The recursive case, as described in Item 1, is the most interesting one. First, note the use of new as a hiding operator. This means that the functionality and window may refer to the same hidden variables, and that thus the window may impose conditions on what values the hidden variables should become bound to. We assume that the renaming $\mathsf{new}_A \circ \mathsf{new}_{X_1} \circ \mathsf{new}_{Z_1}$ maps A to the hidden variable H_1, X_1 to H_2 and Z_1 to H_3. The window imposes that X must become bound to $[H_1 \mid H_2]$ and that Z must become bound to $[H_1 \mid H_3]$. The window also contains requirements imposed by the recursive call $\mathrm{merge}(H_2, Y, H_3)$. Exactly what these requirements are depends on the tail s' of the oracle. The functionality says that when X is a list with at least one element, the agent may bind X to $[H_1 \mid H_2]$ and Z to $[H_1 \mid H_3]$. The functionality also includes the functionality of the recursive call. $\square$

9.7 Correctness

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

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

Definition 9.7.1 For $\langle f, w\rangle \in$ BUNDLE and $a \in \mathsf{A}$, let

$$\alpha\,\langle f,w\rangle = \{t \mid \mathrm{fn}\, t \sqsubseteq \mathrm{E}_H(f), f(\lim t) \in w, \lim t \in \exists_H(w)\}, \quad \text{and}$$
$$\alpha a = \{\alpha\,\langle f,w\rangle \mid s \text{ an infinite oracle and } a\ s = \langle f,w\rangle\}.$$

□

The correctness of the fixpoint semantics is stated as follows.

Theorem 9.7.2 (Correctness) Let A be an agent, Π a program and σ the least fixed point of $\mathsf{P}[\![\Pi]\!]$. We have $\mathcal{A}_\Pi[\![A]\!] = \alpha(\mathsf{E}[\![A]\!]\sigma)$.

The rest of this section (Section 9.7) is devoted to the proof of the correctness theorem.

9.7.1 Soundness

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

Theorem 9.7.3 (Soundness) Let A be an agent, Π a program and σ the least fixed point of $\mathsf{P}[\![\Pi]\!]$. If $t \in \mathcal{A}_\Pi[\![A]\!]$ it follows that $t \in \alpha(\mathsf{E}[\![A]\!]\sigma)$.

The theorem follows from the three lemmas below, whose proofs are given in Section 9.11

Lemma 9.7.4 Let t be a trace such that $t \in \mathcal{A}_\Pi[\![A(s)]\!]$. We have $\mathrm{fn}(t) \sqsubseteq \mathrm{E}_H f$, where $f = \mathsf{F}(\mathsf{E}[\![A]\!]\sigma s)$, and σ is the least fixpoint of $\mathsf{P}[\![\Pi]\!]$.

Lemma 9.7.5 Let t be a trace such that $t \in \mathcal{A}_\Pi[\![A(s)]\!]$. We have $\lim(t) \in \mathrm{E}_H w$, where σ is the least fixpoint of $\mathsf{P}[\![\Pi]\!]$, and $w = \mathsf{W}(\mathsf{E}[\![A]\!]\sigma s)$.

Lemma 9.7.6 Let t be a trace such that $t \in \mathcal{A}_\Pi[\![A(s)]\!]$. We have $f(\lim t) \in w$, where $\langle f, w\rangle = \mathsf{E}[\![A]\!]\sigma s$.

9.7.2 Completeness

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

Theorem 9.7.7 (Completeness) Let $t \in \alpha(\mathsf{E}[\![A]\!]\sigma s)$, for an agent A, and an infinite oracle s. It follows that $t \in \mathcal{A}_\Pi[\![A(s)]\!]$.

This section (Section 9.7.2) is devoted to the proof of the theorem. To avoid repetitions we will assume a program Π, an agent A and an infinite oracle s. We also assume that $\sigma_0 = \bot$, $\sigma_{n+1} = \mathsf{P}[\![\Pi]\!]\sigma_n$, for $n \in \omega$, and $\sigma = \sqcup_{n\in\omega}\sigma_n$.

In some constraint systems there are constraints that cannot be expressed as limits of an ω-chain of finite constraints. If a constraint c cannot be expressed as the limit of an ω-chain of finite constraints it is obvious that there cannot be a trace with limit c. In other words, if we have a constraint c in some window w and want to construct a trace with limit c, we should first make sure that c is the limit of some ω-chain of finite constraints.

Since we will reason about constraints of this type, it is appropriate to give the concept a name and state some of its properties.

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

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

Proposition 9.7.8 Let A be an agent, σ an ω-approximable environment and s an oracle. Then $\mathsf{F}(\mathsf{E}[\![A]\!]\sigma s)$ is ω-approximable.

For a program Π, the function $\mathsf{P}[\![\Pi]\!]$ is ω-approximable. The least fixpoint of $\mathsf{P}[\![\Pi]\!]$ is ω-approximable.

Proposition 9.7.9 Let c and d be ω-approximable constraints. Let f be a closure operator over constraints such that $f \sqsupseteq (c \to d)$. Then there is a trace t such that $\lim t = d$ and $(c \to d) \sqsubseteq \operatorname{fn} t \sqsubseteq f$.

Proof. We have $c = \sqcup_{i\in\omega} c_i$ and $d = \sqcup_{i\in\omega} d_i$, for $c_0, \ldots$ and $d_0, \ldots$ finite constraints.

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

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

We want to show that $\mathrm{fn}\, t \sqsupseteq (c \to d)$. It is sufficient to show that $(\mathrm{fn}\, t) \bigcap (\bot \to c) \sqsupseteq (\bot \to d)$. Clearly, $(\mathrm{fn}\, t) \bigcap (\bot \to c) \sqsupseteq (\bot \to \lim t)$. If we can show that $\lim t \sqsupseteq d$ we are done.

Suppose that $\lim t \not\sqsupseteq d$. There is a least j such that $\lim t \not\sqsupseteq d_j$. We have $f(c) \sqsupseteq d \sqsupseteq d_j$ and thus a least k such that $f(c_k) \sqsupseteq d_j$. Let i be the maximum of j and k. We have $f(c_i) \sqsupseteq d_j$, and thus $f(e_{2i}) \sqsupseteq d_j$, and by the construction above $e_{2i+1} \sqsupseteq d_j$. We have arrived at a contradiction and conclude that $\mathrm{fn}\, t \sqsupseteq (c \to d)$. □

As a step toward the proof of completeness, we show the following proposition which essentially implies that if the trace is given by the fixpoint semantics, we can construct traces $t_0, t_1, \ldots$ which are all given by the operational semantics and are such that $\lim t_i = \lim t$, for all $i \in \omega$, and $\mathrm{fn}\, t = \bigcap_{i \in \omega} t_i$.

Proposition 9.7.10 Let $n \in \omega$ and $\langle f, w \rangle = \mathsf{E}[\![A]\!]\sigma_n s$. Let $d \sqsubseteq e \sqsubseteq c$ be constraints independent of hidden variables such that $(d \to e) \sqsubseteq f$, $c = \exists_H (f\, c)$ and $f\, c \in w$. It follows that there is an $A(s)$-computation with a corresponding trace t such that $\mathrm{fn}\, t \sqsupseteq (d \to e)$ and $\lim t = c$.

Proof. The proof is given in Section 9.11. □

Lemma 9.7.11 Let t be a trace such that $\mathrm{fn}\, t \sqsubseteq \mathsf{F}(\mathsf{E}[\![A]\!]\sigma s)$. 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 $(v(t)_i \to v(t)_{i+1}) \sqsubseteq \mathrm{fn}\, t$. For $i \in r(t)$, let t_i be a trace with $\lim t_i = \lim t$ and $\mathrm{fn}\, t_i = (v(t)_i \to v(t)_{i+1})$. For a fixed i, there is an $n \in \omega$ such that $\mathrm{fn}\, t_i \sqsubseteq \mathsf{F}(\mathsf{E}[\![A]\!]\sigma_n s)$. By Proposition 9.7.10 it follows that there is a computation t'_i with functionality stronger than t_i and limit equal to that of t_i. By Theorem 8.7.1 there is a computation t' such that $\mathrm{fn}\, t' \sqsupseteq \mathrm{fn}\, t'_i$, for each i, and thus $\mathrm{fn}\, t \sqsubseteq \mathrm{fn}\, t'$. □

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

Proposition 9.7.12 Suppose that $A_0(s_0) : d_0 \longrightarrow^* A(s) : d$ and there is an ω-approximable constraint $e \in \mathsf{W}(\mathsf{E}[\![A_0]\!]\sigma s_0)$. There is an initially fair $A(s)$-computation with limit e.

Proof. The proof is given in Section 9.11. □

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

Lemma 9.7.13 Let c be an ω-approximable constraint. If $c \in \mathsf{W}(\mathsf{E}[\![A]\!]\sigma s)$, it follows that there is a fair $A(s)$-computation with limit c.

Proof. We will construct a family of initially fair chains $(L_i^k)_{i\in\omega}$ such that $L_0^0 = [A(s) : d]$, for some d, and each chain has limit c. Further, we make sure that $L_0^k \rightsquigarrow L_0^{k+1}$ and for each k and i there is a k' such that $L_i^k \rightsquigarrow L_0^{k'}$. We will show that given this, the sequence $(L_0^k)_{k\in\omega}$ is a fair chain with limit c.

Let $(L_i^0)_{i\in\omega}$ be an initially fair $A(s)$-computation with limit c. For $k \geq 0$, let $L_0^{k+1} = L_0^k * L_1^{k-1} * \ldots * L_{k-1}^1 * L_k^0$. and let $\{L_i^k\}_{i>0}$ be such that $(L_i^k)_{i\in\omega}$ is initially fair.

It is easy to verify that the family $\{(L_i^k)_{i\in\omega}\}_{k\in\omega}$ satisfies the properties mentioned above. It follows immediately that $(L_0^k)_{k\in\omega}$ is a chain. To verify that $(L_0^k)_{k\in\omega}$ is fair, consider a suffix $(L_0^k)_{k\geq m}$. Since $(L_i^m)_{i\in\omega}$ is initially fair, and because of Lemma 8.7.9, we find that the suffix must be initially fair. It follows that $(L_0^k)_{k\in\omega}$ is a fair chain. □

We are now ready to give the proof of Theorem 9.7.7. Recall that t is a trace such that $\lim t \in \mathsf{W}(\mathsf{E}[\![A]\!]\sigma s)$ and $\mathrm{fn}\, t \sqsubseteq \mathsf{F}(\mathsf{E}[\![A]\!]\sigma s)$ and we want to show the existence of a trace $t' \in \mathcal{O}_\Pi[\![A]\!]$ such that t is a subtrace of t'.
Proof. (Theorem 9.7.7) By Lemma 9.7.13 there is a fair $A(s)$-computation with limit $\lim t$. By Lemma 9.7.11 there is an $A(s)$-computation with functionality greater than or equal to $\mathrm{fn}\, t$ and limit equal to $\lim t$. By Theorem 8.7.1 the two computations can be combined into a fair computation with functionality at least as strong as $\mathrm{fn}\, t$ and limit equal to $\lim t$. □

9.8 Category-theoretic semantics

In this section, we will use a powerdomain construction by Lehmann [46, 47] to devise a fixpoint semantics which is more abstract than the oracle-based fixpoint semantics. This powerdomain construction has previously been used by Abramsky [2], Panangaden and Russel [61], Nyström and Jonsson [58], and de Boer, Di Piero and Palamidessi [24] to give the fixpoint semantics of various forms of nondeterministic programming languages.

Lehmann's construction relies on a special type of categories called ω-categories.

Definition 9.8.1 An *ω-category* is a category which has an initial object and in which all ω-chains have colimits.

An *ω-functors* is a functor which preserves colimits of ω-chains. □

It is easy to see that a cpo or a complete lattice can also be seen as a ω-category, and that a continuous function over a cpo or complete lattice is an ω-functor over the corresponding category.

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

Definition 9.8.2 Assuming a cpo $(D, \sqsubseteq)$, the objects and arrows in the corresponding powerdomain $CP(D)$ are as follows.

The objects are multisets over D. To represent the multisets, we assume some set of *tags*, about which we make no assumptions, except that they exist in sufficient number to represent the multisets we are interested in. We now represent each multiset over D by a set of pairs x_γ, where $x \in D$ and γ is some tag.

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 $x \leq y$. We can view the arrow r as representing a function $r^{-1} : B \to A$, satisfying $r^{-1}y \sqsubseteq y$, 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 $f_i^{-1}(\sqcup_{i \in \omega}(x_i)) = x_i$, for $\sqcup_{i \in \omega}(x_i) \in S$.

9.8.1 Constructions

In the category $CP(D)$, the *product* $A \times B$ is simply the disjoint union. Given objects A and B, let $C = \{x_{\langle \gamma, 1 \rangle} \mid x_\gamma \in A\} \cup \{y_{\langle \gamma, 2 \rangle} \mid y_\gamma \in B\}$. Let $r_1^{-1}(x_\gamma) = x_{\langle \gamma, 1 \rangle}$, for $x_\gamma \in A$, and, similarly, $r_2^{-1}(y_\gamma) = y_{\langle \gamma, 2 \rangle}$, 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 $\times$ is an ω-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 $\uplus$ for the product.

The dual notion of product, *coproduct*, will also be used in the category-theoretic fixpoint semantics. If D is a lattice, and A and B are objects of $CP(D)$, the coproduct $C = A + B$ can be formed by

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

The arrow $r_1 : A \to C$ is given by $r_1^{-1} z_{\langle \gamma_1, \gamma_2 \rangle} = x_{\gamma_1}$, for $x_{\gamma_1} \in A$, $y_{\gamma_2} \in B\}$ 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 $\mathbf{A}^S$ for the category $\mathbf{A}_1 \times \ldots \times \mathbf{A}_n$. If $a \in S$ is the kth element of S let index_a be a functor $\mathrm{index}_a : \mathbf{A}^S \to \mathbf{A}$ such that for objects $\mathrm{index}_a \langle A_1, \ldots, A_n\rangle = A_k$ and for arrows $\mathrm{index}_a \langle f_1, \ldots, f_n\rangle = f_k$

We will use the category product when modelling environments, i.e., mappings from the domain of names to some semantic domain.

We will also need the following result regarding the construction of ω-functors, which is by Lehmann [47]. Given a continuous function $f : D_1 \to D_2$, define the operation $\hat{f}$ as follows. For an object $A \in D_1$, let

$$\hat{f}(A) = \{y_\gamma \mid y = f(x), x_\gamma \in A\},$$

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

$$\hat{f}(r) = \{\langle y_\gamma, y'_{\gamma'}\rangle \mid \langle x_\gamma, x'_{\gamma'}\rangle \in r, y = f(x), y' = f(x')\}.$$

Proposition 9.8.3 Let $f : D_1 \to D_2$ be a continuous function. Then $\hat{f} : CP(D_1) \to CP(D_2)$ is an ω-functor.

9.8.2 The Powerdomain of Trace Bundles

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

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

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

Disjoint union One operation that comes with the categorical powerdomain is the disjoint union $\biguplus$. 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 ω-functors F_1, F_2 : Env $\to$ Proc we can construct an ω-functor $\biguplus(F_1, F_2)$: Env $\to$ 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, and not distinguish explicitly between $\uplus$ 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

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

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 $\{F_j : \mathrm{Env} \to \mathrm{Proc}\}_{j \in I}$ and returns a new functor $\sum_{j \in I} : \mathrm{Env} \to \mathrm{Proc}$ 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 $\mathrm{select}(c) : \mathrm{Proc} \to \mathrm{Proc}$ 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.

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

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

$$\mathrm{unless}(c_1, \ldots, c_n) = \{\langle \mathbf{id}, \mathcal{U} \setminus (\{c_1\}^u \cup \ldots \cup \{c_n\}^u) \rangle\}$$

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), i.e., let

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

For each agent A define a functor $\mathcal{E}[\![A]\!] : \mathrm{Proc}^{\mathcal{N}} \to \mathrm{Proc}$ according to the following equations

$$\begin{aligned}
\mathcal{E}[\![c]\!] &= (\!(c)\!) \\
\mathcal{E}[\![\textstyle\bigwedge_{j\in I} A^j]\!] &= \textstyle\sum_{j\in I}(\theta_j \circ \mathcal{E}[\![A^j]\!]) \\
\mathcal{E}[\![\square_{1\le k\le n}\, c_k \Rightarrow A_k]\!] &= (\textstyle\biguplus_{1\le k\le n} \mathrm{select}(c_k) \circ \mathcal{E}[\![A_k]\!]) \\
&\quad \uplus \mathrm{unless}(c_1, \ldots, c_n) \\
\mathcal{E}[\![\exists_X A]\!] &= \mathsf{new}_X \circ \mathcal{E}[\![A]\!] \\
\mathcal{E}[\![P(X)]\!] &= \{\alpha \to X\} \circ \mathrm{index}_P
\end{aligned}$$

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

$$\mathcal{P}[\![\Pi]\!] = \langle \{Y \to \alpha\} \circ \mathcal{E}[\![A_p]\!] \rangle_{p\in\mathcal{N}},$$

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

Figure 9.2: The categorical fixpoint semantics

The categorical fixpoint semantics

The categorical fixpoint semantics is given in Figure 9.2.

9.9 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. One obvious way of doing this would be to define a mapping from the semantic domain in the oracle based fixpoint to the powerdomain of trace bundles, but it turns out that we need a slightly more complex construction. The strategy we adopt is to devise a fixpoint semantics which is half-way between the oracle fixpoint semantics and the categorical fixpoint semantics, with the intention that the relationship to both fixpoint semantics should be clear.

9.9.1 Augmenting the oracle semantics

Recall that the semantic domain for agents in the oracle based fixpoint semantics, A, is the set of functions from oracles to trace bundles, that is, pairs consisting 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, so we might define an abstraction operator that maps each element $a : \textsc{oracle} \rightarrow \textsc{bundle}$ to a multiset

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

However, this construction is problematic since the minimal member of A, which is $\lambda s.\,\langle \mathbf{id}, \mathcal{U}\rangle$, would be mapped to a process object where each trace bundle had an infinite multiplicity, instead of being mapped to the initial object of the category of processes. Also, the introduction of such multiplicities of elements appears to be rather unnatural.

For a given agent, we must find a set of oracles that is sufficient to generate trace bundles corresponding to all possible computation paths, but which still bears some relationship to the choices performed by the agent. One possible way to determine the set of oracles is to examine the result of the oracle-semantics. However, there does not appear to be any continuous operation that will accomplish this. Instead we define a semantic function $\mathcal{D}$ which will provide us, for each agent, with a set of oracles which are sufficient to determine the set of traces generated.

Recall that in Section 8.3 an operation $\odot$ was defined which for a family of oracles $\{s_n\}_{n\in\omega}$ produced an oracle s such that $\pi_n s = s_n$, for $n \in \omega$. We will use the operation here, with the assumption that $I = \omega$.

For a partial order P, say that a set $S \subseteq P$ is *anticonsistent* if no two elements in S have an upper bound in P, i.e., for all $x, y \in S$ such that $x, y \leq z$, for some $z \in P$ we have $x = y$

Let AC be the set of anticonsistent subsets of $\textsc{oracle}$. For $a, b \in \mathrm{AC}$, say that $a \sqsubseteq b$ if $a^u \supseteq b$ in $\textsc{oracle}$. It turns out that AC forms a cpo under this ordering with $\{\epsilon\}$ as least element. Now we can define $\mathcal{D}[\![A]\!] : \mathrm{AC}^N \rightarrow \mathrm{AC}$ inductively, for each agent A.

$$\begin{aligned}
\mathcal{D}[\![c]\!]\delta &= \{\epsilon\}\\
\mathcal{D}[\![\textstyle\bigwedge_{j\in I} A^j]\!]\delta &= \{\odot_{j\in I} s_j \mid \\
&\qquad s_j \in \mathcal{D}[\![A^j]\!]\delta, \text{ for } j \in I\}\\
\mathcal{D}[\![(c_1 \Rightarrow A_1 \,[]\, \ldots \,[]\, c_n \Rightarrow A_n)]\!]\delta &= \{k.s \mid 1 \leq k \leq n, s \in \mathcal{D}[\![A_k]\!]\delta\}\\
&\quad \cup\{0\}\\
\mathcal{D}[\![\exists_X A]\!]\delta &= \mathcal{D}[\![A]\!]\delta\\
\mathcal{D}[\![p(X)]\!]\delta &= \delta p
\end{aligned}$$

For a program Π we can now define a functor $\mathcal{Q}[\![\Pi]\!] : \mathrm{AC}^{\mathcal{N}} \rightarrow \mathrm{AC}^{\mathcal{N}}$ according to the equation

$$\mathcal{Q}[\![\Pi]\!]\delta p = \mathcal{D}[\![A]\!]\delta,$$

where the definition of p in Π has the form $p(X) :: A$. It should be clear that for a program Π and agent A, and δ the least fixpoint of $\mathcal{Q}[\![\Pi]\!]$, the set $S = \mathcal{D}[\![A]\!]\delta$ is sufficient to determine the set of traces generated by the oracle semantics. For example, for the agent

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

the set of oracles needed to produce all traces is $\mathcal{D}[\![A]\!]\delta = \{0, 1, 2\}$.

9.9.2 An intermediary category

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

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 $P_1 \xrightarrow{r} P_2$ of *CP*(BUNDLE) is a reverse mapping $r^{-1} : P_2 \to P_1$ mapping each tagged element of P_2 to an element of P_1 which is smaller or equal.

Definition 9.9.1 Let INTER be the subcategory 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 oracles, and each member of an object has a unique tag, and the tags of members of an object form an anticonsistent 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 $s_1 \sqsubseteq s_2$.

□

For an oracle s and an object P of 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, INTER is a subcategory of *CP*(BUNDLE). It should also be clear that that INTER has an initial element given by $\langle \mathbf{id}, \mathcal{U} \rangle_\epsilon$ which is also an initial element of *CP*(BUNDLE). For an ω-chain

$$P_0 \xrightarrow{r_0} P_1 \xrightarrow{r_1} P_2 \xrightarrow{r_2} \ldots$$

the colimit can be given by

$$P = \{(\sqcup_{i \in \omega} P_i(s_i))_s \mid s_i \in P_i, \text{ for } i \in \omega \text{ and } s = \sqcup_{i \in \omega} s_i\}.$$

Clearly, an ω-colimit in the category INTER coincides with the corresponding colimit in *CP*(BUNDLE).

So INTER is a sub-ω-category of *CP*(BUNDLE).

9.9.3 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 is of course the natural way to define operations over multisets, 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

$$(\!|c|\!) = \{\langle \bot \to c, \{c\}^u \rangle_\epsilon\}.$$

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 union of multi-sets.

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

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

The semantic equation for selection in Figure 9.2 should thus be read

$$\mathcal{E}[\![\mathop{[\,]}_{k \leq n} c_k \Rightarrow A_k]\!] = \biguplus_{0 \leq k \leq n} P_k,$$

where $P_0 = \text{unless}(c_1, \ldots, c_n)$, and $P_k = \text{select}(c_k) \circ \mathcal{E}[\![A_k]\!]$, for $1 \leq k \leq n$.

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).

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

Selections It is straight-forward to refine the functor $\mathrm{select}(c) : \mathrm{Proc} \to \mathrm{Proc}$ to a functor over the intermediate category. Let

$$\mathrm{select}(c)P = \{\langle c \to f, \{c\}^u \cap w\rangle_s \mid \langle f, w\rangle_s \in P\}.$$

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

Existential quantification In the category-theoretic semantics, existential quantification is obtained by applying the function 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, i.e., let

$$\mathsf{new}_X P = \{\langle \mathsf{new}_X f, \mathsf{new}_X w\rangle_s \mid \langle f, w\rangle_s \in P\}.$$

Intermediate fixpoint semantics

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

Relation with the oracle semantics

For a given program and agent, 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 $\mathcal{D}$ and $\mathcal{Q}$ provides us, for each agent, 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, and the corresponding set of oracles is S, we can give the set of trace bundles as the set $\{as \mid s \in S\}$. The corresponding mapping to the intermediate category is

$$\alpha_i(a, S) = \{(as)_s \mid s \in S\}.$$

9.10 Concluding remarks

In this chapter we presented two fixpoint semantics for concurrent constraint programming. Both semantics take into account infinite computations and fairness between processes.

The first semantics has a conventional lattice-theoretic domain and is rather straight-forward, and it easy to see exactly which aspects of the semantics that makes it less than fully abstract (that is, the use of oracles and that the values of local variables are part of the semantics).

Other methods for giving the semantics of non-deterministic concurrent languages, 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 and might make the oracle semantics useful in the analysis of concurrent programs.

The second fixpoint semantics, the categorical semantics, is what remains when we remove the oracles from the oracle 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.

9.11 Proofs of Chapter 9

Proof of Proposition 9.3.2

We want to show that a renaming θ which is injective when seen as a function over variables is also injective when seen as a function over constraints. We assume that θ is injective, that is, $\theta c = \theta d$, 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 $\phi \in \theta d$, thus there is a $\phi' \in d$ such that $\theta\phi \preceq \theta\phi'$. Let V be an assignment such that $V \models \phi'$. Since θ is injective there is a renaming θ' such that $\theta' \circ \theta = \mathbf{id}$. It follows that with $V' = V \circ \theta'$ we have $V' \circ \theta \models \phi'$ (since $V' \circ \theta = V \circ \theta' \circ \theta = V$) and thus $V' \models \theta\phi'$. Since $\theta\phi \preceq \theta\phi'$ we have $V' \models \theta\phi$ and $V \models \phi$. We have $\phi \preceq \phi'$, and since 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 $c = d$.

Proof of Proposition 9.3.3

Proof of Item 1 To prove that $\exists_H \circ \mathsf{new}_X = \exists_H \circ \exists_X$, we will show that an arbitrary constraint c and variable assignment V, we have $V \models \exists_H(\exists_X c)$ iff $V \models \exists_H(\mathsf{new}_X c)$.

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

It is now easy to see that $V'' \circ \mathsf{new}_X$ satisfies the condition for V' above. Thus, if $V \models \exists_H(\mathsf{new}_X c)$ 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 Y, $V''(\mathsf{new}_X Y) = V'(Y)$, for hidden variables Y, and $V''(\mathsf{new}_X X) = V'(X)$. An assignment V'' that satisfies these conditions also satisfies $V'' \circ \mathsf{new}_X = V''$, thus $V'' \models \mathsf{new}_X c$, and it follows that $V \models \exists_H(\mathsf{new}_X c)$.

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

Proof of Item 2 To show that $\mathsf{new}_X(\exists_X c) \sqsubseteq c$, suppose that $V \models c$. Since $(V \circ \mathsf{new}_X)Y = V(Y)$, for visible variables Y distinct from X we have $V \circ \mathsf{new}_X \models \exists_X c$. It follows immediately that $V \models \mathsf{new}_X(\exists_X c)$.

Proof of Item 3 (We show that $\mathsf{new}_X^{-1} c = \exists_X c$.) ($\sqsupseteq$) By Item 2 we have $\mathsf{new}_X(\exists_X c) \sqsubseteq c$. By applying new_X^{-1} on both sides we find that $\exists_X c \sqsubseteq \mathsf{new}_X^{-1} c$.

($\sqsubseteq$) Note that $\mathsf{new}_X^{-1} c = \bigsqcup\{d \mid \mathsf{new}_X d \sqsubseteq x\}$. To show that $\mathsf{new}_X^{-1} c \sqsubseteq \exists_X c$, it is sufficient to show that for all d such that $\mathsf{new}_X d \sqsubseteq c$, we have $d \sqsubseteq \exists_X c$.

Let d be such that $\mathsf{new}_X d \sqsubseteq c$. 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 $V'(Y) = V(Y)$, for all variables $Y \neq X$. Let V'' be an assignment such that $V''(\mathsf{new}_X Y) = V(Y)$, for all variables Y, 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 $V'' \models \mathsf{new}_X d$, and $V'' \circ \mathsf{new}_X \models d$. Since $V'' \circ \mathsf{new}_X = V$, we have $V \models d$, sor $d \sqsubseteq \exists_X d$.

Proof of Item 4 To show that $\theta_j c = c$, first suppose that $V \models c$, for an assignment V. Since $\theta_j(X) = X$, 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 $\theta_j^{-1}c = c$, we use the result we just obtained, $\theta_j c = c$, and apply the inverse projection θ_j^{-1} to both sides.

Proof of Item 5 In the proof of $\theta_j^{-1} \circ \theta_k = \exists_H$, first note that since $\theta_j^{-1} \sqsupseteq \exists_H$ and $\theta_k \sqsupseteq \exists_H$ we have immediately $\theta_j^{-1} \circ \theta_k \sqsupseteq \exists_H$. To show that $\theta_j^{-1} \circ \theta_k \sqsubseteq \exists_H$, first note that $\theta_j^{-1}(\theta_k c) = \bigsqcup\{d \mid \theta_j d \sqsubseteq \theta_k c\}$, for arbitrary constraints c. Let c be fixed.

Suppose that d is such that $\theta_j d \sqsubseteq \theta_k c$. We will show that $d \sqsubseteq \exists_H c$. Let V be such that $V \models \exists_H c$. It follows that $V' \models c$, for some V' such that $V(X) = V'(X)$, for visible variables X. It is possible to construct a variable assignment V'' such that $V''(X) = V'(X) = V(X)$, for visible variables X, and $V'' \circ \theta_j = V$ and $V'' \circ \theta_k = V'$. To see how this is possible, remember that the sets $\theta_j H$ and θ_k are disjoint, so the values assigned by V and V' to hidden variables cannot interfere. Now, since $V'' \circ \theta_k = V'$, we have $V'' \circ \theta_k \models c$ and thus $V'' \models \theta_k c$. By assumption, this implies that $V'' \models \theta_j d$. Thus $V'' \circ \theta_j \models d$. Since $V'' \circ \theta_j = V$, we have $V \models d$ and we are done.

Proof of Proposition 9.3.4

($\Leftarrow$) Suppose that c is a fixpoint of $\theta \circ f \circ \theta^{-1} \sqcup \mathbf{id}$. It follows that $(\theta \circ f \circ \theta^{-1})c \sqsubseteq c$. Thus $\theta^{-1}((\theta \circ f \circ \theta^{-1})c) \sqsubseteq \theta^{-1}c$, and $f(\theta^{-1}c) \sqsubseteq \theta^{-1}c$. Since f is a closure operator it follows that $\theta^{-1}c$ is a fixpoint of f and thus $c \in \theta f$.

($\Rightarrow$) Suppose that $c \in \theta f$. It follows that $\theta^{-1}c = f(\theta^{-1}c)$, and thus $\theta(f(\theta^{-1}c)) = \theta(\theta^{-1}c) \sqsubseteq c$. We have $\theta(f(\theta^{-1}c)) \sqcup c = c$, and thus c is a fixpoint of $\theta \circ f \circ \theta^{-1} \sqcup \mathbf{id}$.

Proof of Proposition 9.3.5

($\sqsupseteq$) Let $c \in \mathrm{E}_H(\mathsf{new}_X S)$. There is a constraint d such that $\exists_H d = \exists_H c$ and $\mathsf{new}_X^{-1} d \in S$. Let $e = \mathsf{new}_X^{-1} d$. To prove that $c \in \mathrm{E}_H(\mathrm{E}_X S)$, it is sufficient to find a constraint d' such that $\exists_H c = \exists_H d'$ and $\exists_X d' = \exists_X e$. Let $d' = \exists_H c \sqcup \exists_X e$. First, note that $\exists_H d' = \exists_H(\exists_H c \sqcup \exists_X e) = \exists_H c \sqcup \exists_H(\exists_X e)$. Since $\exists_H(\exists_X e) = \exists_H(\mathsf{new}_X e) \sqsubseteq \exists_H d = \exists_H c$ we have $\exists_H c = \exists_H d'$. Second, we have $\exists_X d' = \exists_H(\exists_X c) \sqcup (\exists_X e)$. Since

$$\begin{aligned}
\exists_H(\exists_X c) &= \exists_X(\exists_H d)\\
&= \exists_X(\exists_X(\exists_H d))\\
&= \exists_X(\mathsf{new}_X^{-1}(\exists_H d))\\
&\sqsubseteq \exists_X(\mathsf{new}_X^{-1} d)\\
&= \exists_X e,
\end{aligned}$$

it follows that $\exists_X d' = \exists_X e$.

($\sqsubseteq$) Let $c \in \mathrm{E}_H(\mathrm{E}_X S)$. There is a constraint $e \in S$ such that $\exists_H(\exists_X c) = \exists_H(\exists_X e)$. We would like to find a constraint d such that $\exists_H d = \exists_H c$ and $\mathsf{new}_X^{-1} d \in S$. Let $d = \exists_H c \sqcup \mathsf{new}_X e$. First we see that $\exists_H d = \exists_H c \sqcup \exists_H(\mathsf{new}_X e)$. But $\exists_H(\mathsf{new}_X e) = \exists_H(\exists_x e) = \exists_H(\exists_X c) \sqsubseteq \exists_H c$, so we can conclude that $\exists_H d = \exists_H c$. Second, applying the new_X^{-1} function gives us $\mathsf{new}_X^{-1} d = \mathsf{new}_X^{-1}(\exists_H d) \sqcup \mathsf{new}_X^{-1}(\mathsf{new}_X e) = \mathsf{new}_X^{-1}(\exists_H d) \sqcup e$. We see that $\mathsf{new}_X^{-1}(\exists_H d) = \exists_X(\exists_H d) = \exists_H(\exists_X e) \sqsubseteq e$, thus $\mathsf{new}_X^{-1} d = e$.

Proof of Proposition 9.3.6

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

$$\begin{aligned}(\theta_j f_j)d &= (\theta_j f_j)\left(\textstyle\bigsqcup_{k\in I}(\theta_k f_k)c\right)\\ &= (\theta_j \circ f_j \circ \theta_j^{-1})\left(\textstyle\bigsqcup_{k\in I}(\theta_k \circ f_k \circ \theta_k^{-1})c\right)\\ &= (\theta_j \circ f_j)\left(\textstyle\bigsqcup_{k\in I}(\theta_j^{-1} \circ \theta_k \circ f_k)c\right)\end{aligned}$$

Now, note that for $j = k$ we have $(\theta_j^{-1} \circ \theta_k \circ f_k)c = f_j c$, and for $j \neq k$ we have $(\theta_j^{-1} \circ \theta_k \circ f_k)c = \exists_H(f_k c) \sqsubseteq \exists_H(fc)$. Since $f_j c \sqsupseteq c = \exists_H(fc) \sqsupseteq \exists_H(f_k c)$, for $k \in I$, it follows that $\left(\bigsqcup_{k\in I}(\theta_j^{-1} \circ \theta_k \circ f_k)c\right) = f_j c$. Thus

$$\begin{aligned}(\theta_j \circ f_j)\left(\textstyle\bigsqcup_{k\in I}(\theta_j^{-1} \circ \theta_k \circ f_k)c\right) &= (\theta_j \circ f_j)(f_j c)\\ &= (\theta_j \circ f_j)c\\ &= (\theta_j f_j)c.\end{aligned}$$

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

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

Proof of Lemma 9.7.4

We will prove the following, by induction on pairs (k, A), under the lexical ordering. Let $f = \mathsf{F}(\mathsf{E}[\![A]\!]\sigma s)$. For any $A(s)$-computation $(A_i : c_i)_{i\in\omega}$, when $A_k : c_k \to A_{k+1} : c_{k+1}$, we have $c_{k+1} \sqsubseteq (\mathrm{E}_H f)c_k$.

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

Suppose $A = \bigwedge_{j\in I} A^j$. Let $\langle f_j, w_j\rangle = \mathsf{E}[\![A^j]\!]\sigma(\pi_j s)$, for $j \in I$. By the computation rules $A_i = \bigwedge_{j\in I} A_i^j$, for all $i \in \omega$, and there is a $j \in I$ such that $A_k^j : c_k \to A_{k+1}^j : c_{k+1}$. By the induction hypothesis we have $c_{k+1} \sqsubseteq (\mathrm{E}_H f_j)c_k$. Since $f \sqsupseteq \theta_j f_j$, and $\mathrm{E}_H(\theta_j f_j) = \mathrm{E}_H f_j$, we have $c_{k+1} \sqsubseteq (\mathrm{E}_H f_j)c_k = (\mathrm{E}_H(\theta_j f_j))c_k \sqsubseteq (\mathrm{E}_H f)c_k$.

If A is a selection, 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 $f' = \mathsf{F}(\mathsf{E}[\![A']\!]\sigma s)$. For each $i \in \omega$, we know that $A_i \exists_X^{d_i} A'_i$, for some d_i and A'_i. By the computation rules we have $A'_k : d_k \sqcup \exists_X(c_k) \longrightarrow A'_{k+1} : d_{k+1}$, where $c_{k+1} = c_k \sqcup \exists_X(d_{k+1})$. For $i \leq k$, we have by the induction hypothesis that $d_{i+1} \sqsubseteq f'(d_i \sqcup \exists_X(c_i))$. We can prove by an inductive argument that $d_{i+1} \sqsubseteq (f' \circ \exists_X)c_k$, for all $i \leq k$. First, $d_0 = \bot$, by the assumption that the local data in an initial configuration is $\bot$. Suppose $d_i \sqsubseteq (f' \circ \exists_X)c_k$. Now we have, when $i \leq k$, that

$$\begin{aligned} d_{i+1} &\sqsubseteq f'(d_i \sqcup \exists_X(c_i)) \\ &\sqsubseteq f'((f' \circ \exists_X)c_k \sqcup \exists_X(c_i)) \\ &= (f' \circ f' \circ \exists_X)c_k \\ &= (f' \circ \exists_X)c_k. \end{aligned}$$

In particular we have $d_{k+1} \sqsubseteq (f' \circ \exists_X)c_k$. It follows that $c_{k+1} \sqsubseteq c_k \sqcup (\exists_X \circ f' \circ \exists_X)c_k = fc_k$. By monotonicity we have $\exists_H(c_{k+1}) \sqsubseteq \exists_H(fc_k)$, and since c_k and c_{k+1} are independent of variables in H, $c_{k+1} \sqsubseteq \exists_H(fc_k) = \exists_H(f(\exists_H c_k)) \sqsubseteq (\mathrm{E}_H f)c_k$.

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

$$\begin{aligned} \mathsf{E}[\![p(X)]\!]\sigma s &= \{\alpha \to X\}(\sigma p s) \\ &= \{\alpha \to X\}(\{Y \to \alpha\}(\mathsf{E}[\![A']\!]\sigma s)) \\ &= \mathsf{E}[\![A_i]\!]\sigma s. \end{aligned}$$

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

Proof of Lemma 9.7.5

Let $\sigma_0 = \lambda s.\lambda P.\langle \mathbf{id}, \mathcal{U}\rangle$, $\sigma_{n+1} = \mathsf{P}[\![\Pi]\!]\sigma_n$, for $n \geq 0$. Let $w_n = \mathsf{W}(\mathsf{E}[\![A]\!]\sigma_n s)$, for $n \geq 0$. Note that $w = \bigcap_{n \in \omega} w_n$, so if $\lim t \notin w$, it follows that $\lim t \notin w_n$, for some n.

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

If A is a tell constraint c it follows immediately from the fairness requirements that any trace $t \in \mathcal{O}_\Pi[\![A]\!]$ must have $\lim t \sqsupseteq c$, i.e., $\lim t \in \{c\}^u = w$, and since $\lim t$ is independent of hidden variables, $\lim t \in \mathrm{E}_H w$.

Suppose $A = \bigwedge_{j\in I} A^j$. It follows by Lemma 4.7.5 that there are traces $t_j \in \mathcal{O}_\Pi[\![A^j]\!]$ such that $t = \bigvee_{j\in I} t_j$. By the induction hypothesis we have $\lim t_j \in \mathrm{E}_H(w^j)$, where $w^j = \mathsf{W}(\mathsf{E}[\![A^j]\!]\sigma_n s)$. Since $\lim t = \lim t_j$, for all $j \in I$, $\lim t \in \bigcap_{j\in I} \mathrm{E}_H(w^j) = \mathrm{E}_H(\bigcap_{j\in I} w^j) = \mathrm{E}_H w$.

Suppose A is a selection $(d_1 \Rightarrow A_0 \;[]\; \ldots \;[]\; d_m \Rightarrow A_m)$, and $s = k.s'$. Suppose $1 \le k \le m$. 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 $A_l = A_k$, i.e, the kth alternative must eventually be selected. By the computation rules we find that this cannot happen unless $c_{l'} \sqsupseteq d_k$, for some $l' \le l$. Now, consider the $A_k(s')$-computation $(A_i : c_i)_{i\ge l}$. By the induction hypothesis, we know that the limit of this computation, $\bigsqcup_{i\ge l} c_i$, lies in $\mathrm{E}_H(\mathsf{W}(\mathsf{E}[\![A_k]\!]\sigma_n s'))$. It follows that $\lim t \in \mathrm{E}_H w$.

Suppose A is a selection $(d_1 \Rightarrow A_0 \;[]\; \ldots \;[]\; d_m \Rightarrow A_m)$ and $s = 0.s'$. By the fairness requirement, it holds for each store c_i that $c_i \not\sqsupseteq d_k$, for $k \le m$. Thus we have $\lim t = \sqcup_{i\in\omega} c_i \not\sqsupseteq d_k$, for $k \le m$, and it follows that $\lim t \in \mathsf{W}(\mathsf{E}[\![A]\!]\sigma_n s)$.

Suppose $A = \exists_X A'$. We know that t is of the form $(\exists_X^{d_i} A'_i : c_i)_{i\in\omega}$ and that $t' \in \mathcal{O}_\Pi[\![A']\!]$, where $t' = (A'_i : d_i \sqcup (\exists_X c_i))_{i\in\omega}$.

It follows by the assumption on A that $d \sqcup (\exists_X c) \in \mathrm{E}_H w'_n$, where $w'_n = \mathsf{W}(\mathsf{E}[\![A']\!]\sigma_n s)$, and $c = \sqcup_{i\in\omega} c_i$, and $d = \sqcup_{i\in\omega} d_i$. By the computation rules we know that $c \sqsupseteq \exists_X d$. It follows that $\exists_X c = \exists_X(c \sqcup \exists_X d) = \exists_X(d \sqcup \exists_X c) \in \mathrm{E}_X(\mathrm{E}_H w'_n)$. Thus, $c \in \mathrm{E}_X(\mathrm{E}_H w'_n) = \mathrm{E}_H(\mathrm{E}_X w'_n) = \mathrm{E}_H(\mathsf{new}_X w'_n) = \mathrm{E}_H(w_n)$.

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

Proof of Lemma 9.7.6

Let $\sigma_0 = \lambda s.\lambda p.\, \langle \mathbf{id}, \mathcal{U}\rangle$, and $\sigma_{n+1} = \mathsf{P}[\![\Pi]\!]\sigma_n$, for $n \in \omega$. Let $\sigma = \bigsqcup_{n\in\omega} \sigma_n$. We will show by induction on pairs (n, A) that whenever $t \in \mathcal{A}_\Pi[\![A(s)]\!]$, we have $f\,c \in \mathsf{W}(\mathsf{E}[\![A]\!]\sigma_n s)$, where $c = \lim t$.

If A is a tell constraint d, it follows by fairness that $\lim t \sqsupseteq d$. Thus $\lim t \in \{d\}^u = w$.

If A is a conjunction $\bigwedge_{j\in I} A^j$ we have $t_j \in \mathcal{A}_\Pi[\![A^j]\!]$ such that $\lim t_j = \lim t$, for all $j \in I$. With $\langle f_j, w_j\rangle = \mathsf{E}[\![A^j]\!]\sigma_n(\pi_j s)$ we have, by the induction hypothesis, $f_j c \in w_j$. Since $w = \bigcap_{j\in I} \theta_j\, w_j$ we want to show that $f\,c \in \theta_k\, w_k$, i.e., that $\theta_k^{-1}(f\,c) \in w_k$, for all $k \in I$. By Proposition 9.3.6 $f\,c = \bigsqcup_{j\in I}(\theta_j f_j)c = \bigsqcup_{j\in I} \theta_j(f_j c)$. Let k be fixed in I. We have $\theta_k^{-1}(f\,c) =$

$\theta_k^{-1}(\bigsqcup_{j\in I}\theta_j(f_j c)) = \bigsqcup_{j\in I}\theta_k^{-1}(\theta_j(f_j c))$, and since $\theta_k^{-1}\circ\theta_j = \exists_H$, for $j \neq k$, it follows that $\bigsqcup_{j\in I}\theta_k^{-1}(\theta_j(f_j c)) = f_k c \in w_k$.

Suppose A is a selection $(d_1 \Rightarrow B_1 \,[]\, \ldots \,[]\, d_m \Rightarrow B_m)$. If s begins with a 0 it follows from fairness that $\lim t \not\sqsupseteq d_k$, for $1 \le k \le m$, thus $f = \mathbf{id}$ and $f(\lim t) = \lim t \in w$. If $s = k.s'$, where $1 \le k \le m$, it follows that $f = d_k \to f'$ and $w = \{d_k\}^u \cap w'$ where $\langle f', w'\rangle = \mathsf{E}[\![B_k]\!]\sigma_n s'$. By the induction hypothesis $f'\,c \in w'$. By fairness the kth branch of the selection must be chosen, and in order for this to happen the ask constraint d_k must be entailed by the store. Thus, $c \sqsupseteq d_k$ so $f\,c = f'\,c \in w'$. Since $c \in \{d_k\}^u$ we are done.

Suppose $A = \exists_X A'$. We have $t' \in \mathcal{A}_\Pi[\![A']\!]$, and with $c' = \lim t'$, we also have $\exists_X c' = \exists_X c$, $\mathrm{E}_X(\mathrm{fn}\, t') = \mathrm{E}_X(\mathrm{fn}\, t)$ and $(\mathrm{fn}\, t')(\exists_X c) = c'$. With $\langle f', w'\rangle = \mathsf{E}[\![A^j]\!]\sigma_n$ we have $f'\,c' \in w'$. We want to show $f\,c \in w$ which is true iff $\mathsf{new}_X^{-1}(f\,c) \in w'$. Now,

$$\begin{aligned}
\mathsf{new}_X^{-1}(f\,c) &= \mathsf{new}_X^{-1}((\mathsf{new}_X f')c)\\
&= \mathsf{new}_X^{-1}(c \sqcup \mathsf{new}_X(f'(\mathsf{new}_X^{-1}c)))\\
&= (\exists_X c) \sqcup f'(\exists_X c)\\
&= f'(\exists_X c)\\
&\in w'.
\end{aligned}$$

If A is a call $p(X)$ and the definition of p is of the form $p(Y) :: A'$ it follows that t is also a trace of $\mathcal{A}_\Pi[\![[X/Y]A'(s)]\!]$. According to the fixpoint semantics $\mathsf{E}[\![A]\!]\sigma_n = \{\alpha \to X\}(\sigma_n ps) = \{\alpha \to X\}(\{Y \to \alpha\}\mathsf{E}[\![A']\!]\sigma_{n-1}s)$. Clearly, $\{\alpha \to X\}(\{Y \to \alpha\}\mathsf{E}[\![A']\!]\sigma_{n-1}s) = \mathsf{E}[\![[X/Y]A']\!]\sigma_{n-1}s$. By induction hypothesis $(\{\alpha \to X\}(\{Y \to \alpha\}f))c \in \{\alpha \to X\}(\{Y \to \alpha\}w)$, from which follows that $f\,c \in w$.

Proof of Proposition 9.7.10

The proof is by induction on pairs (n, A) under the lexical ordering. We only consider the cases where A is a conjunction or an existential quantifier.

Consider the agent $\bigwedge_{j\in I} A^j$. Let $\langle f_j, w_j\rangle = \mathsf{E}[\![A^j]\!]\sigma_n(\pi_j s)$.

Let $q : \omega \to I$ be a mapping such that for each $j \in I$, $q(i) = j$ infinitely often. Let $d_0 = d$ and $d_{i+1} = \exists_H(f_{q(i)}d_i)$, for $i \in \omega$. Let $e' = \bigsqcup_{i\in\omega} d_i$. If $c \sqsupseteq d_i$, it follows that $c = \exists_H(fc) \sqsupseteq \exists_H(f_k d_i)$, for any k, thus $c \sqsupseteq e'$. On the other hand, since $\exists_H(fd) \sqsupseteq e$, we have $e \sqsubseteq \exists_H(fd) \sqsubseteq \exists_H(\bigcap_{j\in I}(\theta_j f_j)d)$. For $k \in I$, let $e'_k = (\theta_k f_k)e'$. Note that $\exists_H e'_k = e'$. Since for $j \neq k$ we have

$$\begin{aligned}
((\theta_j f_j)\circ(\theta_k f_k))e' &= (\theta_j f_j)e'_k\\
&= e'_k \sqcup (\theta_j \circ f_j \circ \theta_j^{-1})e'_k\\
&= e'_k \sqcup (\theta_j \circ f_j)e'\\
&= e'_k \sqcup e'_j,
\end{aligned}$$

it is clear that $\exists_H(\bigcap_{j\in I}(\theta_j f_j)d) = e'$. Thus $e \sqsubseteq e'$.

To be able to apply the induction hypothesis we must show that $f_k c \in w_k$, for all $k \in I$. By Proposition 9.3.6 we have $fc = \bigsqcup_{j\in I}(\theta_j f_j)c$. Let $k \in I$ be fixed. We have $fc \in (\theta_k w_k)$. Thus, $fc = \theta_k c_k$, for some $c_k \in w_k$. So $\theta_k^{-1}(fc) = c_k$, and since $\theta_k^{-1}(fc) = \theta_k^{-1}(\bigsqcup_{j\in I}(\theta_j f_j)c) = f_k c$ 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 $A^j(\pi_j s)$-computation (where $j = q(i)$) with trace t_i, such that $\mathrm{fn}\, t_i \sqsupseteq d_i \to d_{i+1}$ and $\lim t_i = c$. By Theorem 8.7.1 there is an $A(s)$-computation with trace t that satisfies the theorem.

Suppose $A = \exists_X A'$. Let $\langle f', w'\rangle = \mathsf{E}[\![\exists_X A']\!]\sigma_n s$. Let $d' = \exists_X d$ and $e' = (f' \circ \exists_X)d$. Clearly $(d' \to e') \sqsubseteq f'$. Let $c' = \exists_X c$. To be able to apply the induction hypothesis we must show that $f'c' \in w'$. First note that $\mathsf{new}_X^{-1}(fc) \in w'$. We have

$$\begin{aligned}
\mathsf{new}_X^{-1}(fc) &= \mathsf{new}_X^{-1}((\mathsf{new}_X f')c)\\
&= \mathsf{new}_X^{-1}((\mathsf{new}_X \circ f' \circ \mathsf{new}_X^{-1} \sqcup \mathbf{id})c)\\
&= (\mathsf{new}_X^{-1} \circ \mathsf{new}_X \circ f' \circ \mathsf{new}_X^{-1} \circ \exists_H)c \sqcup (\mathsf{new}_X^{-1} \circ \exists_H)c\\
&= f'(\exists_X c) \sqcup \exists_X c\\
&= f'c'
\end{aligned}$$

By the induction hypothesis there is an $A'(s)$-computation with trace t' such that $\lim t' = c'$ and $\mathrm{fn}\, t' \sqsupseteq (d' \to e')$. For $i \in r(t')$ let $d_i = \forall_X(v(t')_i)$, i.e., the least d_i such that $\exists_X(d_i) \sqsupseteq v(t')_i$, and $e_i = \exists_X(v(t')_{i+1})$. We can for each $i \in r(t')$ construct an $A(s)$-computation with trace t_i such that $\mathrm{fn}\, t_i \sqsupseteq (d_i \to e_i)$ and $\lim t_i = c'$. It follows from Theorem 8.7.1 that there is a computation t with $\lim t = c$ and

$$\mathrm{fn}\, t \sqsupseteq \bigcap_{i\in r(u)} \mathrm{fn}\, t_i = \mathrm{E}_X(\mathrm{fn}\, u) \sqsupseteq (d \to e).$$

Proof of Proposition 9.7.12

Note that $\mathsf{W}(\mathsf{E}[\![A_0]\!]\sigma s_0) \cap \{d_0\}^u = \mathsf{W}(\mathsf{E}[\![A]\!]\sigma s) \cap \{d\}^u$, if $A_0(s_0) : d_0 \longrightarrow^* A(s) : d$.

The proposition is proved by induction on pairs (A, n), under the lexical ordering, 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 $\mathcal{K}(\mathcal{U})$. such that $e = \bigcup_{i\in\omega} e_i$.

Suppose A is a tell constraint c. It follows that $c \sqsubseteq e$. 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, an oracle s_1, a constraint d_1 and $n' < n$ such that $A_0(s_0) : d_0 \longrightarrow^{n-n'} A_1(s_1) : d_1$ and $A_1(s_1) : d_1 \longrightarrow^{n'} A(s) : d$. It follows that we can apply the induction hypothesis (since $(A, n') < (A, n)$

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 $A_0^j(\pi_j s_0) : d \longrightarrow^n A^j(\pi_j s) : d$. 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 $(c_1 \Rightarrow B_1 \mathbin{[]} \ldots \mathbin{[]} c_m \Rightarrow B_m)$. If the oracle s begins with a 0 it follows that $c_k \not\sqsubseteq e$, 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, where $1 \leq k \leq m$, it follows that $c_k \sqsubseteq e$. Let $l \in \omega$ be such that $e_l \sqsupseteq c_k$. We have an initially fair computation $(A(s) : e_0, A(s) : e_l, B_k(s') : e_l, B_k(s') : e_{l+1}, \ldots)$, where s' is the tail of s.

Suppose A is an existential quantification $\exists_X^c A'$. 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 $e' = \mathsf{F}(\mathsf{E}[\![A']\!]\sigma s)(\exists_X e)$. By Proposition 9.7.9 there is a trace t such that $\lim t = e'$ and $\operatorname{fn} t \sqsupseteq (\exists_X(e) \to e')$ and $\operatorname{fn} t \sqsubseteq f$. By the induction hypothesis there is an initially fair $A'(s)$-computation with limit e'. By Lemma 9.7.11 there is an $A'(s)$-computation with functionality stronger than $\operatorname{fn} t$ and limit e'. By Theorem 8.7.1 it follows that these two computations may be combined into an initially fair $A'(s)$-computation with limit e' and functionality stronger than $\operatorname{fn} t$. Let B be the agent $\bigwedge_{i\in\omega} \exists_X(e_i)$, i.e., a conjunction of agents in which each agent is a tell constraint, 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', and r the set of computation steps performed by A' in the computation.

We will now give c_i and d_i, for $i \in \omega$, such that $\exists_X^{c_i} A'_i : d_i$ is an initially fair computation with limit e.

Let $c_0 = d_0 = \bot$. If $i \in r$, let $d_{i+1} = d_i \sqcup \exists_X(w_{i+1})$ and $c_{i+1} = w_{i+1}$. If $i \notin r$, let $d_{i+1} = d_i \sqcup e_k$ if the ith step was performed by executing the kth factor of B. Let $d_{i+1} = d_i$ otherwise. Let $c_{i+1} = c_i$. It is straightforward 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

$$(A(s) : e_0, A'[Y/X](s) : e_0, A'[Y/X](s) : e_1, \ldots)$$

is an initially fair computation.

Chapter 10

Concluding Remarks

In the introductory chapter I stated three goals of the thesis. The first goal was that the semantics should consider a formalism with the expressiveness of a normal programming language, with data structures, recursion, process creation and a flexible communication model. The second goal was that the semantics should consider infinite computations, and the third goal was to strive for semantic models that are based on abstract decencies between the input and output of processes and to avoid models based on communication events. In this chapter I discuss how these goals were fulfilled, and point out some further directions of research.

Concurrent constraint programming turned out to be an appropriate formalism for the exploration of concurrency. Ccp is on one hand a powerful concurrent programming language, and has a relatively straight-forward formal definition. Also, ccp is parametric in the sense that the constraint system can be any system of logic formulas (that satisfies some simple requirements); this means that results about the semantics of ccp have a very wide range of potential applications.

There were, however, a few early decisions in my formal definition of ccp that made the further developments unnecessarily complicated. The mechanism for parameter passing made it more difficult than necessary to reason formally about the semantics of calls. The way of defining calls also made it necessary to require that formulas such as $X = Y$ should be a part of the constraint system; a different formalisation of the operational semantics of calls would have made it possible to greatly simplify the definition of constraint systems. Also, the operational semantics of the existential quantifier made the proof of finite confluence unnecessarily complicated.

One goal of this thesis is to give denotational semantics for ccp. To discuss how well this goal was fulfilled, we must first ask ourselves what constitutes a denotational semantics. It is generally agreed that a denotational semantics should be compositional and give meaning to loop constructs and recursive programs by fixpoint iteration. However, a 'semantics' whose domains are based on the syntactic forms of programs and programs fragments, with infinite unfoldings to describe the meaning of recursive programs, would satisfy these requirements, even though it is clearly unacceptable. One way of ruling out semantics models based on the syntactic form is the requirement that a denotational semantics should be fully abstract, but for some languages (such as ccp) there is no fully abstract fixpoint semantics. One can also wonder whether full abstraction is a reasonable criteria even for languages where there is a fully abstract fixpoint semantics. For the deterministic data flow language considered by Kahn (and also for deterministic ccp) there is a very elegant fully abstract fixpoint semantics whose domain is the continuous functions from input to output histories, but the trace-based semantics of non-deterministic data flow is by Russell's proof also a fully abstract semantics for deterministic data flow. Yet I am sure that no-one will disagree with me when I claim that Kahn's semantics is preferable to a trace-based semantics. Why is Kahn's semantics better? The difference lies in the choice of domains. A semantics whose domain consists of the continuous function over some space is clearly more attractive than a semantics based on uncountable sets of infinite sequences of events. A space of continuous functions is a well-understood mathematical concept, but if we represent the same process using the set of possible traces the properties of the process disappear behind the peculiarities of the trace representation. The conclusion I draw is that besides the useful requirement of full abstraction the formulation of the semantics also matters; we should strive for formulations of the semantics where the domain and the composition operations are given at the highest possible level of abstraction.

The fully abstract semantics satisfies this requirement partially in that the composition operators are given in terms of the functionality and limit of traces, but the domain of the fully abstract semantics is the set of all subtrace-closed sets of traces, which is not the most elegant construction one can imagine.

One reason to consider fully abstract semantics is that the set of algebraic identities satisfied by a fully abstract semantics will be the identities satisfied by any semantics. In the case of ccp, the fully abstract semantics turns out to satisfy the axioms of intuitionistic linear algebra, an algebra which was defined to capture the properties of intuitionistic linear logic. It was also interesting to note that selection could be expressed using other operations of intuitionistic linear logic. It is difficult to judge the importance of these results, but to me the match between ccp and intuitionistic linear algebra seems too strong to be dismissed as a coincidence.

Two proofs of full abstraction were given. The first relied on the use of infinite conjunctions to provide an appropriate context and as it could be argued that this context is not a realistic program I gave a second proof in which the context was finite but depended on an infinite input. It is worthwhile to ask whether it really is necessary to introduce infinite information in the context. After all, the set of finite agents is countable (if we make some reasonable assumptions on the constraint system) so one would expect that a countable set of contexts should be sufficient to distinguish agents with differing behaviour. Even though the set of traces of an agent are in general uncountable it may be possible to select a set of 'countable' traces so that if two agents differ in behaviour, there is some computable trace that one agent can exhibit but not the other. What is interesting here is not (only) the prospect of finding a slightly more general proof of full abstraction, but also the idea that a there may be a countable set of traces that can capture the infinite behaviour of agents.

The idea behind the oracle semantics is to record the non-deterministic choices made by an agent. This simple idea made it possible to show a generalised confluence property that involves infinite sets of infinite computations.

The fixpoint semantics based on oracles is a fairly straight-forward construction, where the semantics of an agent is given in terms of domain constructions and operations which are of a high level of abstraction. However, the fixpoint semantics fails to be fully abstract. This is not surprising in view of the results presented in the thesis, but even with the negative results in mind it is still disappointing to discover that the oracles do not provide sufficient information to allow a fixpoint semantics. However, the fixpoint semantics obtained is nevertheless a fairly compact construction based on simple concepts, and I hope it will prove useful in the future research on the semantics of concurrency.

One possible application of the fixpoint semantics is in the static analysis of concurrent programs. An analysis based on the fixpoint semantics could of course only provide information about the external behaviour of agents, in contrast to other static analyses, which typically are intended to provide information about internal aspects of computations. Information about the external behaviour of a process could be used in various compiler optimisations, and also as a program development tool.

References

[1] Samson Abramsky. Experiments, powerdomains and fully abstract models for applicative multiprogramming. In *Proc. Foundations of Computation Theory*, volume 158 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 1983.

[2] Samson Abramsky. Semantic foundations of applicative multiprogramming. In *Proc. ICALP '83*, volume 154 of *Lecture Notes in Computer Science*, pages 1–14. Springer-Verlag, 1983.

[3] Samson Abramsky and Stephen Vickers. Quantales, observational logic, and process semantics. Technical Report DOC 90/1, Imperial College, Dept. of Computing, January 1990.

[4] K. R. Apt and G. D. Plotkin. Countable nondeterminism and random assignment. *Journal of the ACM*, 33(4):724–767, October 1986. First published as technical report, Department of Computer Science, University of Edinburgh, 1982.

[5] Joe Armstrong, Robert Virding, and Mike Williams. *Concurent Programming in ERLANG*. Prentice-Hall, 1993.

[6] Geoff Barret. The fixed point theory of unbounded non-determinism. *Formal Aspects of Computing*, 3:110–128, 1991.

[7] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60:109–137, 1984.

[8] Per Brinch Hansen. The nucleus of a multiprogrammed system. *Communications of the ACM*, 13(4):238–250, April 1970.

[9] J. Dean Brock and William B. Ackerman. Scenarios: a model of nondeterminate computation. In Diaz and Ramos, editors, *Formalization of Programming Concepts, LNCS 107*, volume 107 of *Lecture Notes in Computer Science*, pages 252–259. Springer-Verlag, 1981.

[10] Jarvis Dean Brock. *A Formal Model of Non-determinate Dataflow Computation*. PhD thesis, Massachusetts Institute of Technology, 1983.

[11] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.

[12] S. D. Brookes and A. W. Roscoe. An improved failures model for communicating processes. In Brookes, Roscoe, and Winskel, editors, *Proc. Seminar on Concurrency, 1984*, volume 197 of *Lecture Notes in Computer Science*, pages 268–280. Springer-Verlag, 1985.

[13] Stephen Brookes. Full abstraction for a shared variable parallel language. In *Proc. 8th IEEE Int. Symp. on Logic in Computer Science*, pages 98–109, 1993.

[14] Manfred Broy. A theory for nondeterminism, parallelism, communication, and concurrency. *Theoretical Computer Science*, 45:1–61, 1986.

[15] J. M. Cadiou and J. J. Lévy. Mechanizable proofs about parallel processes. In *Switching and Automata Theory Symposium*, volume 14, pages 34–48. IEEE, 1973.

[16] Björn Carlson. *An Approximation Theory for Constraint Logic Programs.* Thesis for the Degree of Licientiate of Philosophy, Uppsala University, 1991.

[17] Keith Clark and Steve Gregory. A relational language for parallel programming. In *ACM conference on Functional Programming and computer architecture*, pages 171–178. ACM, 1981.

[18] Keith Clark and Steve Gregory. Parlog: Parallel programming in logic. *ACM Trans. on Programming Languages and Systems*, 8(1):1–49, 1986.

[19] William Douglas Clinger. *Foundations of Actor Semantics.* PhD thesis, MIT, May 1981.

[20] Rina S. Cohen and Arie Y. Gold. Theory of ω-languages. I: Characterizations of ω-context-free languages. *Journal of Computer and Systems Sciences*, 15:169–184, 1977.

[21] J. W. de Bakker and J. N. Kok. Uniform abstraction. atomicity and contractions in the comparative semantics of concurrent Prolog. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 347–355. ICOT, 1988.

[22] F. S. de Boer, J. N. Kok, C. Palamidessi, and J. J. M. M. Rutten. The failure of failures in a paradigm for asynchronous communication. In *Proceedings of CONCUR'91*, number 527 in Lecture Notes in Computer Science, pages 111–126. Springer-Verlag, 1991.

[23] Frank S. de Boer and Catuscia Palamidessi. A fully abstract model for concurrent constraint programming. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, number 493 in Lecture Notes in Computer Science, pages 296–319. Springer-Verlag, 1991.

[24] Frank S. de Boer, Alessandra Di Pierro, and Catuscia Palamidessi. Nondeterminism and infinite computations in constraint programming. *Theoretical Computer Science*, 151:36–78, 1995.

[25] Jack B. Dennis. First version of a data flow procedure language. Technical Report 61, Laboratory for Computer Science, Massachusetts Institute of Technology, May 1975. First published in B. Robinet (ed), *Programming Symposium: Proceedings Colloque sur la Programmation*, Lecture Notes in Computer Science 19, April 1974., 362-376.

[26] E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, New York, 1968. First published as Technical Report EWD-123, Technological University, Eindhoven, 1965.

[27] Ian Foster and Stephen Taylor. Strand: A practical programming tool. In Ewing L. Lusk and Ross A. Overbeek, editors, *North American Conference on Logic Programming*, pages 497–512. MIT Press, 1989.

[28] Ian Foster and Stephen Taylor. *Strand: New Concepts in Parallel Programming.* Prentice-Hall, 1989.

[29] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislowe, and D. S. Scott. *A Compendium of Continuous Lattices.* Springer-Verlag, 1980.

[30] Leon Henkin, J. Donald Monk, and Alfred Tarski. *Cylindric Algebras*, volume 1. North-Holland, 1971.

[31] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–676, August 1978.

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

[33] Radha Jagadeesan, Keshav Pingali, and Prakash Panangaden. A fully abstract semantics for a functional programming language with logic variables. *ACM Trans. on Programming Languages and Systems*, 13(4):577–625, October 1991. Also in Proceedings of the IEEE Symposium on Logic in Computer Science 1989.

[34] Bengt Jonsson. A model and proof system for asynchronous networks. In *Proc. 4th ACM Symp. on Principles of Distributed Computing*, pages 49–58, Minaki, Canada, 1985.

[35] Bengt Jonsson. A fully abstract trace model for dataflow and asynchronous networks. *Distributed Computing*, 7:197–212, 1994.

[36] Mark B. Josephs. Receptive process theory. *Acta Informatica*, 29:17–31, 1992.

[37] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proceedings of IFIP Congress*, pages 471–475. North-Holland, 1974.

[38] Peter Kearney and John Staples. An extensional fixed-point semantics for nondeterministic data flow. *Theoretical Computer Science*, 91:129–179, 1991.

[39] Robert M. Keller. Denotational models for parallel programs with indeterminate operators. In Neuhold, editor, *Formal Descriptions of Programming Concepts*, pages 337–366. North-Holland, 1978.

[40] Joost N. Kok. A fully abstract semantics for data flow nets. In *Proc. PARLE*, volume 259 of *Lecture Notes in Computer Science*, pages 351–368. Springer-Verlag, 1987.

[41] Paul R. Kosinski. A straight-forward denotational semantics for nondeterminate data flow programs. In *Proc. 5th ACM Symp. on Principles of Programming Languages*, pages 214–219, 1978.

[42] Paul Roman Kosinski. *Denotational semantics of determinate and Nondeterminate Data Flow Programs*. PhD thesis, MIT, May 1979.

[43] Marta Kwiatkowska. Infinite behaviour and fairness in concurrent constraint programming. In *Semantics: Foundations and Applications*, volume 666 of *Lecture Notes in Computer Science*, pages 348–383. Springer-Verlag, 1992.

[44] P. J. Landin. A correspondence between ALGOL 60 and Church's lambda notation: Part I. *Communications of the ACM*, 8(2):89–101, February 1965.

[45] F. William Lawvere. Adjointness in foundations. *Dialectica*, 23(3/4):281–296, 1969.

[46] Daniel J. Lehmann. Categories for fixed-point semantics. In *17th Annual Symposium on Foundations of Computer Science*, pages 122–126, 1976.

[47] Daniel J. Lehmann. *Categories for Fixed-point Semantics.* PhD thesis, Hebrew University of Jerusalem, 1976.

[48] M. J. Maher. Logic semantics for a class of committed-choice programs. In *4th International Conference on Logic Programming*, pages 858–876. MIT Press, 1987.

[49] Kim Marriott and Martin Odersky. A confluent calculus for concurrent constraint programming with guarded choice. In *Proceedings of 1st Conference on Principles and Practice of Constraint Programming*, volume 976 of *Lecture Notes in Computer Science*, pages 310–327, Cassis, France, September 1995. Springer-Verlag.

[50] David May. Occam. *SIGPLAN Notices*, 18(4):69–79, April 1983.

[51] John McCarthy. A basis for a mathematical theory of computation. In P. Brafford and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, 1967. An earlier version was presented at the Western Joint Computer Conference, May 1961.

[52] Nax Paul Mendler, Prakash Panangaden, P. J. Scott, and R. A. G. Seely. A logical view of concurrent constraint programming. *Nordic Journal of Computing*, 2(2):181–220, Summer 1995.

[53] Robin Milner. Processes: A mathematical model of computing agents. In *in Logic Colloquium 1973*, pages 157–173. North-Holland, 1973.

[54] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[55] E. Morenoff and J. B. McLean. Inter-program communications, program string structures, and buffer files. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 175–183, 1967.

[56] M. Nivat. Infinite words, infinite trees, infinite computations. In J.W. de Bakker and J. van Leeuven, editors, *Foundations of Computer Science III*, Mathematical Centre Tracts 109, pages 3–52. Matematisch Centrum, Amsterdam, 1981.

[57] Sven-Olof Nyström. Control structures for Guarded Horn Clauses. In *Fifth International Conference Symposium on Logic Programming*, pages 1351–1370, 1988.

[58] Sven-Olof Nyström and Bengt Jonsson. Indeterminate concurrent constraint programming: a fixpoint semantics for non-terminating computations. In *Proceedings of the 1993 International Logic Programming Symposium*, pages 335–352. MIT Press, 1993.

[59] H. Ono. Phase structures and quantales - a semantical study of logics without structural rules. October 1990, Lecture delivered at the conference *Logics with restricted structural rules*, University of Tübingen. Cited in [79].

[60] Erik Palmgren. Denotational semantics of constraint logic programs—a nonstandard approach. In B. Mayoh, E. Tyugu, and J. Penjam, editors, *Constraint Programming*, NATO ASI Series F, pages 261–288. Springer-Verlag, 1994.

[61] Prakash Panangaden and James R. Russell. A category-theoretic semantics for unbounded nondeterminacy. In *Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 319–332. Springer-Verlag, 1990.

[62] Prakash Panangaden and Vasant Shanbhogue. The expressive power of indeterminate dataflow primitives. *Information and Computation*, 98:99–131, 1992.

[63] D. Park. The 'fairness' problem and nondeterministic computing networks. In de Bakker and van Leeuwen, editors, *Foundations of Computer Science IV, Part 2*, pages 133–161, Amsterdam, 1983. Mathematical Centre Tracts 159.

[64] David Park. On the semantics of fair parallelism. In *Abstract Software Specifications, Copenhagen Winter School 1979*, volume 86 of *Lecture Notes in Computer Science*, pages 504–526. Springer-Verlag, 1980.

[65] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark, 1981.

[66] Axel Poigné. Context-free languages of infinite words as least fixpoints. In *Proc. Int. Conf. on fundamentals on Computation Theory, Szeged, Hungary*, volume 117 of *Lecture Notes in Computer Science*, pages 301–310. Springer-Verlag, 1981.

[67] A. W. Roscoe. Unbounded non-determinism in CSP. *Journal of Logic and Computation*, 3(2):131–172, 1993.

[68] James R. Russell. Full abstraction for nondeterministic dataflow networks. In *Proc.* 30^{th} *Annual Symp. Foundations of Computer Science*, pages 170–177, 1989.

[69] James R. Russell. On oraclizable networks and Kahn's principle. In *Proc.* 17^{th} *ACM Symp. on Principles of Programming Languages*, pages 320–328, 1990.

[70] Vijay A. Saraswat. Problems with Concurrent Prolog. Technical Report CMU-CS-86-100, Carnegie-Mellon University, Computer Science Department, 1986,1985.

[71] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In *Proc. 18^{th} ACM Symp. on Principles of Programming Languages*, 1991.

[72] Dana S. Scott. Domains for denotational semantics. In *ICALP'82*, number 140 in Lecture Notes in Computer Science, pages 577–613. Springer-Verlag, 1982.

[73] Robert A. G. Seely. Hyperdoctrines, natural deduction and the Beck condition. *Zeitschr. f. math. Logik und Grundlagen d. Math.*, 29:505–542, 1983.

[74] Ehud Shapiro. Concurrent Prolog: A progress report. *IEEE Computer*, 19(8):44–58, August 1986.

[75] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, September 1989.

[76] Ehud Y. Shapiro. A subset of Concurrent Prolog and its interpreter. Technical Report 003, Institute for New Generation Computer Technology, Tokyo, 1983.

[77] Eugene W. Stark. A simple generalization of Kahn's principle to indeterminate dataflow networks. Technical report, State University of New York, Stony Brook, July 1990. Also published as extended abstract in *Proceedings of the International BCS-FACS Workshop on Semantics for Concurrency*, Leicester 1990, M. Z. Kwiatkowska, M. W. Shields, R. M. Thomas, (eds.), pp. 157-176, Springer-Verlag.

[78] Allen Stoughton. *Fully abstract models of programming languages.* Pitman, 1988.

[79] A. S. Troelstra. *Lectures on Linear Logic.* Number 29 in CSLI Lecture Notes. Center for the study of language and information, Stanford, 1992.

[80] Kazunori Ueda. Concurrent Prolog re-examined. Technical Report 102, Institute for New Generation Computer Technology, Tokyo, 1985.

[81] Kazunori Ueda. Guarded Horn Clauses. Technical Report 103, Institute for New Generation Computer Technology, Tokyo, 1985.

[82] Kazunori Ueda. *Guarded Horn Clauses.* PhD thesis, University of Tokyo, March 1986.

[83] Glynn Winskel and Mogens Nielsen. Models for concurrency. Technical Report PB-463, Computer Science Department, Aarhus University, 1993. To appear in Handbook of Logic in Computer Science, Oxford University Press.