On Non-determinism, Infinite Computations, Fixpoint Semantics and Full Abstraction

Sven-Olof Nyström

Computing Science Department
Uppsala University,
P.O. Box 311, S-751 05 Uppsala, Sweden
svenolof@csd.uu.se

Abstract
Consider the programming languages which satisfy the following properties.

1. There is some form of non-deterministic choice.
2. The program can generate output,
3. The programming language allows arbitrary recursion, i.e., non-guarded recursion is allowed.
4. The results of infinite computations are considered.

We define a very simple language which satisfies the listed properties, in fact, one can argue that it is minimal with respect to the listed properties. We show that is is not possible to give a fully abstract fixpoint semantics for this language. Due to the simplicity of the proof one can expect that it should be possible to apply the proof to any other programming language which satisfies the four points.

It should be stressed that our results are with respect to fixpoint semantics, so we rule out not only any semantics based on continuous functions, but also any non-continuous semantics based on other types of functions.
1 Introduction

It is well known that the problem of giving a fixpoint semantics for a programming language with non-determinism raises difficulties, especially if one wants to consider infinite computations and wants to avoid redundant information in the semantic model.

Why is it so? In 1974 Kahn [4] presented a denotational semantics for a deterministic concurrent programming language. It is obvious that the semantics is fully abstract, and the treatment of recursion (i.e. fixpoints) and infinite computations is immediate. In contrast, to give a denotational semantics for non-deterministic languages people have been forced to make the choice between either making the model contain too much information, or ignoring the results of infinite computations. The latter option is problematic since we want to be able to address questions relating to fairness and liveness.

1.1 Main Points

Consider the programming languages which satisfy 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 context-free grammars provide a simple model which satisfy 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 also allow words generated by infinite derivations. We thus obtain a simple programming language with the syntax of context-free grammars, and an operational semantics which is close to the standard rules of context-free grammars.

We show that is is not possible to give a fully abstract fixpoint semantics for this language.

Since our language is so simple, it is reasonable to expect that the proof should be applicable to any other programming language which satisfies the four points.

It should be stressed that our results are with respect to fixpoint semantics, so we rule out not only any semantics based on continuous functions, but but also any non-continuous function based on other types of functions.
1.2 Related Work

Apt and Plotkin [2] 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.

Abramsky [1] demonstrated that there is no continuous fully abstract fixpoint for a non-deterministic language, similar to the one we consider.

In contrast, we show that the language we consider does not allow any fully abstract least fixpoint semantics, regardless of whether continuity is required or not. One can easily extend the proof to handle other types of fixpoint semantics, i.e., greatest fixpoints semantics or semantics based on topology or category theory.

The main differences between the language we consider here, and the one Apt and Plotkin examined are that our language allows arbitrary recursion and infinite computations, but not unbounded indeterminism. The language in this paper is also much 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 [4] treats infinite computation and arbitrary recursion but does not allow non-determinism.

Brookes [3] 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 [8] give fully abstract fixpoint semantics for various types of concurrent constraint programming languages. One of the languages under consideration is a non-deterministic language which allows arbitrary recursion. However, only finite computations are considered.

A similar result is by Russel [7], who considers a class of non-deterministic data flow networks. He gives a fully abstract fixpoint semantics but does not consider infinite computations.

1.3 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 most important reasons for giving a denotational semantics.

We have demonstrated that if we consider non-deterministic and non-terminating programs, it is not possible to give a fully abstract fixpoint semantics.

2 A suitable formalism

As mentioned in the introductory chapter, 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.

In the following, we will follow the definition of Nivat [5]. He gave a fixpoint interpretation for grammars in Greibach form. This restriction roughly corresponds to the guarded recursion found in various programming languages. However, we will deal with arbitrary grammars.

2.1 Generation of infinite words

Let $N$ be an infinite set of non-terminals and $T$ 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 $\alpha$ is a finite string over $T \cup N$. A finite string over $T \cup N$ will sometimes be referred to as an agent. For a grammar $G$ and a non-terminal $X$, let $G(X)$ be the set of agents $\alpha$ such that there is a production $X \to \alpha$ in $G$.

For a grammar $G$ we define the relation $\Rightarrow_G$ to be the smallest relation over $(T \cup N)^*$ that satisfies

$$uX\alpha \Rightarrow_G u\beta\alpha,$$

for a string $u \in T^*$, a non-terminal $X$ and strings $\alpha, \beta \in (T \cup N)^*$ such that there is a production $X \to \beta$ in $G$. Let $\Rightarrow_G^*$ be the transitive and reflexive closure of $\Rightarrow_G$. (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 a word $\alpha$ is

$$L(G, \alpha) = \{w \in T^* \mid \alpha \Rightarrow^* w\}.$$
When an infinite derivation $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots$ is considered one can imagine many different definitions on 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 Nivat.

For a string $\alpha \in (T \cup N)^*$, say that $w$ is the largest terminal prefix of $\alpha$ if $w \in T^*$, and $w$ is a prefix of $\alpha$, and every word $w' \in T^*$ which is a prefix of $\alpha$ is also a prefix of $w$.

Given an infinite derivation $\alpha_0 \Rightarrow \alpha_1 \ldots \alpha_n \ldots$ we can construct a chain $w_0 \leq w_1 \leq \ldots \leq w_n \leq \ldots$ such that $w_i$ is the largest terminal prefix of $\alpha_i$, for all $i \in \omega$. We say that $w = \bigvee_{i \in \omega} w_i$ is the generated word and use the notation $\alpha_0 \xrightarrow{G} \omega w$. The $\omega$-language generated by the agent $\alpha$ and the grammar $G$ is then

$$L^\omega(G, \alpha) = \{ w \in T^* | \alpha \xrightarrow{G} \omega w \}.$$ 

For the set of words generated by finite and infinite derivations we write

$$L^\infty(G, \alpha) = L(G, \alpha) \cup L^\omega(G, \alpha).$$

So 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 traces of actions performed by an agent). One might think that it should be a fairly straightforward exercise to give the semantics for the language in the form of a denotational semantics. However, we will show that it is in fact not possible to give a fully abstract fixpoint semantics for the language. To give an intuitive grasp of the the difficulties involved, we ask the reader to consider the following two examples. The program

$$A \rightarrow aA | \epsilon$$

will generate a language consisting of all finite strings $a^*$, and also the infinite string $a^\omega$. A similar program

$$A \rightarrow Aa | \epsilon$$

which only differs from the first program in the first production for the non-terminal $A$, will only generate the finite strings given by the regular expression $a^*$, since the derivation

$$A \Rightarrow Aa \Rightarrow Aaa \Rightarrow Aaaa \ldots$$

has the string $\epsilon$ as largest terminal prefix. One important idea in domain theory is that an infinite amount of information should be approximated by finite pieces of information [9]. This idea conflicts with the behaviour of the two programs above, since they behave identically when only finite computations are considered, but differ in the infinite case.
3 A Fully Abstract and Compositional Semantics

In this section we give a semantics which is both fully abstract and compositional, but not a fixpoint semantics.

Consider the result of a single computation. We shall see in the following example that representing the result as a string is not sufficient if we want a compositional semantics.

\[ A \rightarrow \epsilon \]
\[ B \rightarrow B \]

Both programs generate the empty string; \( A \Rightarrow \epsilon \) in a single step. The infinite computation \( B \Rightarrow B \Rightarrow \ldots \) will also generate the empty string. However, the agents \( Aa \) and \( Ba \) differ in that \( \mathcal{L}^\infty(G, Aa) = \{a\} \) but \( \mathcal{L}^\infty(G, Ba) = \{\epsilon\} \). We need a way to distinguish between finite strings generated by terminating computations, and finite strings generated by non-terminating computations. Poigné [6] solves this by adding to the set of finite and infinite words a new set of words consisting of finite words ending with \( \bot \).

The approach taken here is to represent each word as a function over the infinite words, so that a result \( w \) (necessarily finite) generated by a terminating computation is represented by the function \( \lambda u. w_1 \), and a result \( w \) (finite or infinite) produced by a non-terminating computation is represented by the function \( \lambda u. w \).

Given a set of terminal symbols \( T \), let \( T^\infty \) be the set of functions of the forms \( \lambda u.w_1, \lambda u.w \), and \( \lambda u.v \), where \( w \in T^* \) and \( v \in T^\omega \). It should be clear that \( T^\infty \) forms a cpo under the usual ordering of functions over a cpo, with \( \lambda u. \epsilon \) the least element, and that function composition in \( T^\infty \) corresponds to concatenation of strings.

Given the above definition of \( T^\infty \), we can give the abstract semantics as follows.

\[ A_G[\alpha] = \{ \lambda u.w \mid w \in \mathcal{L}(G, \alpha) \} \cup \{ \lambda v.w \mid w \in \mathcal{L}^\omega(G, \alpha) \} \]

It is straightforward to show that \( A_G \) satisfies the following equations.

\[ A_G[\alpha] = \{ \lambda u.w \} \]
\[ A_G[\epsilon] = \{ \lambda w.w \} \]
\[ A_G[\alpha \beta] = A_G[\alpha] \circ A_G[\beta] \]
\[ A_G[X] = \bigcup_{\alpha \in G(X)} \]

(We extend the usual function composition operator to be defined over sets of functions.)

If we have two agents \( \alpha \) and \( \alpha' \) such that \( A_G[\alpha] \neq A_G[\alpha'] \) we must show that there is a context \( C[\cdot] \) where \( \mathcal{L}^\infty(G, C[\alpha]) \neq \mathcal{L}^\infty(G, C[\alpha']) \).

We can assume that \( \mathcal{L}^\infty(G, \alpha) = \mathcal{L}^\infty(G, \alpha') \), because otherwise the proof is trivial.
Let \( f \in A_G[\alpha] \setminus A_G[\alpha'] \). If \( f = \lambda u.uv, \) for some strings \( u \) and \( v, \) we can conclude that \( \lambda u.v \in A_G[\alpha'] \). Let \( d \) be a terminal symbol that does not occur in \( G. \) Since \( v \in \mathcal{L}(G, \alpha), \) and the string \( v \) thus produced by a finite derivation, it follows that \( uv \in \mathcal{L}(G, \alpha d) \). In the same way, we see that \( v \in \mathcal{L}^\omega(G, \alpha'), \) and that \( v \) is thus produced by an infinite derivation. It follows that \( v \in \mathcal{L}^\omega(G, \alpha'd) \) but \( uv \notin \mathcal{L}^\omega(G, \alpha'd) \) since the \( d \) is 'hidden' by the infinite derivation and thus never added to the generated string. The terminal symbol \( d \) does not occur in \( G \) and can thus not be generated in any other way. If \( f \) is of the form \( \lambda u.u \) we can carry out a similar argument.

We can conclude that the semantics given by the function \( A \) is indeed fully abstract. It also follows that in a fully abstract semantics, two agents \( \alpha \) and \( \alpha' \) should be mapped to the same value if and only if \( \mathcal{L}^\infty(G, \alpha) = \mathcal{L}^\infty(G, \alpha'), \) where \( \alpha \) is some terminal symbol that does not occur in \( G. \) We can generalise this argument for the case where we have two grammars \( G \) and \( G' \), and want to compare the semantics of an agent \( \alpha, \) given the grammar \( G, \) with the semantics of the agent \( \alpha', \) given the grammar \( G'. \)

4 Why no fully abstract fixpoint semantics?

**Definition 1** A fixpoint semantics consists of a cpo \( D \) together with monotone functions

\[
\begin{align*}
\mathcal{E}[\alpha] & : (N \to D) \to D \\
\mathcal{P}[G] & : (N \to D) \to (N \to D)
\end{align*}
\]

such that the following holds.

1. (Correctness) Let \( \sigma \) and \( \sigma' \) be environments such that \( \sigma \) is the least fixpoint of \( \mathcal{P}[G] \) and \( \sigma' \) is the least fixpoint of \( \mathcal{P}[G'] \) for grammars \( G \) and \( G'. \) Whenever \( \mathcal{E}[\alpha][\sigma] = \mathcal{E}[\alpha'][\sigma'] \) it follows that \( \mathcal{L}^\infty(G, \alpha) = \mathcal{L}^\infty(G', \alpha'). \)

2. (Compositionality) Let \( G, G' \) be grammars and \( X, X' \) be non-terms such that \( G(X) = G'(X'). \) It follows that the equality \( \mathcal{P}[G]\sigma X = \mathcal{P}[G']\sigma X' \) holds, for any environment \( \sigma. \)

Motivation. The idea is that the meaning of an agent \( \alpha, \) with respect to a grammar \( G, \) should be given by \( \mathcal{E}[\alpha]\sigma, \) where \( \sigma \) is the environment given by the least fixpoint of \( \mathcal{P}[G]. \) 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. From this follows that given a grammar and an environment, the denotation of that grammar grammar with respect to a particular non-terminal should be determined.
by the productions for that non-terminal. This corresponds to Apt and Plotkin's compositionality requirement.

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 \( \alpha \) and \( \alpha' \) 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 one of the main results of this paper.

**Theorem 2** There is no fully abstract least fixpoint semantics which is both correct and compositional.

Before we turn to the proof, the reader is asked to study the following two grammars.

Grammar \( G_1 \):

\[
\begin{align*}
A & \rightarrow aA \mid \epsilon \mid D \\
B & \rightarrow Aa \mid \epsilon \mid AD \\
D & \rightarrow D
\end{align*}
\]

Grammar \( G_2 \):

\[
\begin{align*}
A & \rightarrow Aa \mid \epsilon \mid AD \\
B & \rightarrow aA \mid \epsilon \mid D \\
D & \rightarrow D
\end{align*}
\]

We can assume that \( d \) is a terminal symbol (which of course does not occur in either \( G_1 \) or \( G_2 \)). 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.

\[
\begin{align*}
\mathcal{L}^\infty(G_1, Ad) &= a^*d \cup a^* \cup a^\omega \\
\mathcal{L}^\infty(G_1, Bd) &= a^*d \cup a^* \cup a^\omega \\
\mathcal{L}^\infty(G_2, Ad) &= a^*d \cup a^* \\
\mathcal{L}^\infty(G_2, Bd) &= a^*d \cup a^*
\end{align*}
\]

Note that \( \mathcal{L}^\infty(G_1, Ad) \) and \( \mathcal{L}^\infty(G_2, Ad) \) only differ in that the infinite string \( a^\omega \) 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 \( \sigma_1 \) be the least fixpoint of the function \( f \) and \( \sigma_2 \) 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 \( \rho \in (N \rightarrow D) \) 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

\[
\sigma'_1 A = g\sigma_1 A \quad \text{(Definition of } \sigma'_1) \\
= f\sigma_1 B \quad \text{(Compositionality argument above)} \\
= f\sigma_1 A \quad \text{(By full abstraction)} \\
= \sigma_1 A \quad \text{(Since } \sigma_1 \text{ is a fixpoint of } f) 
\]

For the non-terminal $B$, we have

\[
\sigma'_1 B = g\sigma_1 B \\
= f\sigma_1 A \quad \text{(By compositionality)} \\
= f\sigma_1 B \quad \text{(By full abstraction)} \\
= \sigma_1 B 
\]

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 $\mathcal{L}_\infty(G_1, Ad) \neq \mathcal{L}_\infty(G_2, Ad)$. $\square$

5 Conclusions

The negative results concerning fully abstract fixpoint semantics 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. Translation to a data flow language with recursion and non-determinism is also straight-forward.

In the full version of the paper, we will give examples of how similar results can be obtained for

1. concurrent constraint programming languages,
2. non-deterministic data flow languages, and
3. an imperative language with shared variables and concurrency.

References


