Denotational Semantics of Evaluation Order in Expressions with Side Effects

Nikolaos S. Papaspyrou
(nickie@softlab.ntua.gr)

National Technical University of Athens
Department of Electrical and Computer Engineering
Division of Computer Science, Software Engineering Laboratory
Polytechnioupoli, 15780 Zografou, Athens, Greece.
Tel. +30-1-7722486, Fax. +30-1-7722519

Abstract

The presence of side effects in even a very simple language of expressions gives rise to a number of semantic questions. The issue of evaluation order becomes a crucial one and, unless a strict order is enforced, the language becomes non-deterministic. In this paper we study the semantics of such a language under a variety of possible evaluation strategies, from simpler to more complex, concluding with unspecified evaluation order, unspecified order of side effects and the mechanism of sequence points that is particular to the ANSI C programming language. In doing so, we adopt the denotational semantics approach and use monads to improve modularity and to reduce the number of changes required for each transition. The result is a better understanding of different evaluation strategies, possibly leading to non-determinism in the presence of side effects, and a unified way of specifying their semantics. Furthermore, a significant step is achieved towards a correct denotational semantics for ANSI C.

1 Introduction

Expressions play a very important role in the vast majority of programming languages. In many languages expressions are *pure*, i.e. their evaluation depends on but may not alter the program state. For such "ideal" languages several theories exist that can be used to prove properties of programs, starting from Hoare's work on axiomatic semantics in the late '60s. Since then, many theoretical treatments of formal semantics have preferred to study languages with pure and deterministic expressions, because their semantics is relatively simple and elegant. By introducing *side effects* in expressions, the semantics becomes significantly more complex. At the same time, several issues arise, related to the *evaluation order*, i.e. the order in which the subparts of an expression are evaluated. If the evaluation order is not strictly defined, the presence of side effects in expressions is a source of non-deterministic behaviour.

The formal semantics of pure expressions can most of the times be easily specified in operational [Henn90], denotational [Stoy77] or axiomatic form [Dijk76]. The introduction of side-effects implies serious complications in the axiomatic semantics but only small ones in the other two forms. Non-determinism caused by unspecified evaluation order can usually be expressed more easily in operational semantics than in denotational semantics. However, both approaches are possible and in this paper the denotational approach is used.

One of the most important drawbacks of classic denotational semantics is its lack of modularity. Small changes in a language's definition often imply a complete rewrite of its formal semantics. The use of category theory and monads [Mogg90] has been proposed as a remedy and has become quite popular in the denotational semantics community. Monad notation is used in this paper and it is demonstrated that, as a result, the semantics is significantly improved in terms of modularity and elegance.

This paper aims at investigating various evaluation strategies for expressions that may generate side effects and providing a unified way of describing their formal semantics. For this reason, a simple strict and impure expression language is studied under various different evaluation strategies, starting from simple left-to-right evaluation and moving on to more complex ones. The language is slightly extended in the process to allow constructs specific to the studied strategies.

The last evaluation strategy that is considered deserves special mention; execution interleaving is allowed and both the order of expression evaluation and the order in which side effects take place are unspecified. The introduction of sequence points and a few additional restrictions makes the example language a subset of the core of the ANSI C programming language. The proposed semantics that develops naturally is sufficient to model this subset correctly and is on its own a significant result. It should be mentioned that the present research was motivated by problems encountered in a bigger project, aiming at a complete denotational semantics for ANSI C.

The structure of this paper is as follows. Section 2 defines the example language and briefly discusses semantic issues related to evaluation order. In Section 3 a denotational semantics is gradually built for four different evaluation strategies. Section 4 shows how our work relates to the literature and Section 5 concludes with some final remarks and directions for future work.

2 Evaluation order

Consider a simple strict expression language with side effects, that will be called ELSE hereafter. The abstract syntax of ELSE is given below. It will be slightly extended in Sections 3.3 and 3.4.

$$E : \mathbf{Expr} \ ::= \ n \mid I \mid I\texttt{=}E \mid E_1\texttt{+}E_2 \mid \texttt{-}E \mid E_1\texttt{,}E_2$$

The language features a single data type (integer), constants, variables, an assignment operator, a binary operator (integer addition), a unary operator (integer negation) and a juxtaposition operator (comma). The informal semantics of ELSE expressions is certainly familiar to most readers. The value of $I\texttt{=}E$ is the value of E and the value of $E_1\texttt{,}E_2$ is the value of E_2.

Apart from the comma operator, which always evaluates its left operand before its right operand, evaluation order of ELSE expressions is left unspecified, for the time being, and various possibilities will be considered in the sequel. It should be clear, however, that this renders the evaluation of ELSE expressions ambiguous, since different evaluation strategies may lead to different results. As an example, consider the simple expression `x=0, x+(x=1)`[1] whose evaluation may result in 1 or 2, depending on whether the (implicit) dereferencing of `x` will take place before or after the assignment. It should also be clear that different evaluation orders are only possible in the presence of operator `+`, since evaluation order is specified for the comma operator and trivially defined for all other constructs.

3 Semantics of ELSE

The denotational semantics of ELSE is given in an abstract form, using monads to improve modularity and elegance of notation. Comprehensive introductions to monads and their use in denotational semantics can be found in [Mogg90, Wadl92, Lian95]. In brief, a monad is a tuple $\langle \mathsf{M}, \mathsf{unit}_\mathsf{M}, *_\mathsf{M}\rangle$, where M is a domain constructor, $\mathsf{unit}_\mathsf{M} : A \to \mathsf{M}(A)$ and $\cdot *_\mathsf{M} \cdot : \mathsf{M}(A) \times (A \to \mathsf{M}(B)) \to \mathsf{M}(B)$ are polymorphic functions for arbitrary domains A and B, satisfying three monad laws.[2] By using different monads, it is possible to derive different flavours of ELSE's denotational semantics without having to change the semantic equations each time.

The flat domain of integer numbers $\mathbf{N}$ is used to denote the values of ELSE's only data type.[3] Bottom element represent abnormal values, mainly run-time errors since non-termination is not possible in ELSE. Domains constructed by monad M denote *computations*, e.g. the domain $\mathsf{M}(\mathbf{N})$ denotes computations returning values from $\mathbf{N}$. The definition of M depends on our notion of computation and it is evident that, in the case of ELSE, computations may read and modify values of variables, i.e. the *state*. The result of $\mathsf{unit}_\mathsf{M}\ v$ is a computation returning v and the result of $m *_\mathsf{M} f$ is the combined computation of m returning v, followed by computation $f\ v$. Evaluation order is thus implicitly specified when using $*_\mathsf{M}$, since the result of the first computation is needed before the second computation can begin.

Before we can proceed to the semantics of ELSE, we have to define the notion of state. Abstractly, the state can be defined as a domain $\mathbf{S}$ together with the following operations,

$$\begin{array}{lcl} s_\circ & : & \mathbf{S} \\ \mathsf{get} & : & \mathbf{Ide} \to \mathbf{S} \to \mathbf{N} \\ \mathsf{put} & : & \mathbf{Ide} \to \mathbf{N} \to \mathbf{S} \to \mathbf{S} \end{array}$$

where $s_\circ$ represents an empty state, i.e. a state where all variables have undefined values, $\mathsf{get}\ I\ s$ reads the value of I in state s, and $\mathsf{put}\ I\ n\ s$ modifies state s by making the value of I equal to n. A simple implementation of state is given below.[4]

$$\begin{array}{lcl} \mathbf{S} & = & \mathbf{Ide} \to \mathbf{N} \oplus \mathbf{U} \\ s_\circ & = & \lambda I.\ \mathbf{inr}\ \mathbf{u} \\ \mathsf{get} & = & \lambda I.\ \lambda s.\ [\mathsf{id}, \bot]\ (s\ I) \\ \mathsf{put} & = & \lambda I.\ \lambda n.\ \lambda s.\ s[I \mapsto \mathbf{inl}\ n] \end{array}$$

The semantic function $[\![\cdot]\!]$ maps an expression E to its denotation $[\![E]\!]$. Denotations of expressions are computations, and since there is only one data type in ELSE, it suffices to take $\mathsf{M}(\mathbf{N})$ as the domain of denotations. The semantic equations for all ELSE constructs are given below,

$$\begin{array}{lcl} [\![n]\!] & = & \mathsf{unit}_\mathsf{M}\ n \\ [\![I]\!] & = & \mathsf{state} *_\mathsf{M} (\lambda s.\ \mathsf{unit}_\mathsf{M}\ (\mathsf{get}\ I\ s)) \\ [\![I\texttt{=}E]\!] & = & [\![E]\!] *_\mathsf{M} (\lambda n.\ \mathsf{update}\ (\mathsf{put}\ I\ n) *_\mathsf{M} \\ & & \quad (\lambda s.\ \mathsf{unit}_\mathsf{M}\ n)) \\ [\![E_1\texttt{+}E_2]\!] & = & ([\![E_1]\!] \bowtie [\![E_2]\!]) *_\mathsf{M} \\ & & \quad (\lambda \langle n_1, n_2\rangle.\ \mathsf{unit}_\mathsf{M}\ (n_1 + n_2)) \\ [\![\texttt{-}E]\!] & = & [\![E]\!] *_\mathsf{M} (\lambda n.\ \mathsf{unit}_\mathsf{M}\ (-n)) \\ [\![E_1\texttt{,}E_2]\!] & = & [\![E_1]\!] *_\mathsf{M} (\lambda n.\ [\![E_2]\!]) \end{array}$$

where, apart from the details of M's implementation, we have yet to define operator $\bowtie$ and functions $\mathsf{update} : (\mathbf{S} \to \mathbf{S}) \to \mathsf{M}(\mathbf{S})$ and $\mathsf{state} : \mathsf{M}(\mathbf{S})$. The last two, as in [Lian95], are used as an interface between computations and the state. The implementation of update depends on monad M and the result of $\mathsf{update}\ f$ is a computation that modifies the state by applying function f and returns the previous state (before the modification). On the other hand, state simply represents a computation returning the present state and can be defined as follows.

$$\mathsf{state} \ = \ \mathsf{update}\ \mathsf{id}$$

Operator $\cdot \bowtie \cdot : \mathsf{M}(A) \times \mathsf{M}(B) \to \mathsf{M}(A \times B)$ is very important in this paper. It "combines" two independent computations into a single one. If $m_a : \mathsf{M}(A)$ and m_b :

[1] In all examples we adopt the same operator precedence and associativity as in C and use parentheses to group expressions. All these details are hidden in the abstract syntax.

[2] This description, although naïve, is sufficient for the purpose of this paper. The notation used is similar to the one used in [Wadl92], with the exception of the bind operator, which is denoted here as $*$.

[3] The choice of the mathematical space used for domains is not very important for such a small language. Pointed cpo's are a reasonable option. The domain notation used in this paper is the same as in [Moss90].

[4] $\mathbf{U}$ is the two-point domain $\{\bot, \mathbf{u}\}$ with $\bot \sqsubseteq \mathbf{u}$. It is used to express values that are not important unless they are abnormal. Also, the function update operator $f[x \mapsto y]$ is strict in all its arguments.

$\mathsf{M}(B)$ are two computations returning values $v_a : A$ and $v_b : B$, then $m_a \bowtie m_b$ is the combined computation returning the pair of values $\langle v_a, v_b \rangle$. The implementation of $\bowtie$ generally depends on the implementation of M and, as is evident from the semantic equations, specifies the order in which expressions are evaluated. The following sections discuss several implementations of $\bowtie$ reflecting various possible evaluation strategies for ELSE. Monad M is carefully defined each time to support the desired evaluation order semantics.

3.1 Left-to-right evaluation

A very simple, common and natural evaluation order is *left-to-right*.[5] In the case of ELSE, left-to-right evaluation specifies that the left operand of + is evaluated completely before the right operand. This is reflected in the following definition of operator $\bowtie$.

$$\begin{aligned} & m_1 \bowtie m_2 \; = \\ & \quad m_1 *_{\mathsf{M}} (\lambda v_1.\, m_2 *_{\mathsf{M}} (\lambda v_2.\, \mathsf{unit}_{\mathsf{M}}\, \langle v_1, v_2 \rangle)) \end{aligned}$$

Before we can have a complete semantics for ELSE, it is necessary to define an appropriate monad M. In this section, we choose a direct semantics approach, making use of an additional monad P as a provision for the following sections. Definitions of M and update are straightforward.

$$\begin{aligned} \mathsf{M}(T) &= \mathbf{S} \to \mathsf{P}(T \times \mathbf{S}) \\ \mathsf{unit}_{\mathsf{M}} &= \lambda v.\, \lambda s.\, \mathsf{unit}_{\mathsf{P}}\, \langle v, s \rangle \\ m *_{\mathsf{M}} f &= \lambda s.\, m\, s *_{\mathsf{P}} (\lambda \langle v', s' \rangle.\, f\, v'\, s') \\ \mathsf{update} &= \lambda f.\, \lambda s.\, \mathsf{unit}_{\mathsf{P}}\, \langle s, f\, s \rangle \end{aligned}$$

For the purpose of this section, the identity monad is a reasonable choice for P, ending up with the conventional direct semantics for ELSE.

$$\begin{aligned} \mathsf{P}(T) &= T \\ \mathsf{unit}_{\mathsf{P}} &= \mathsf{id} \\ p *_{\mathsf{P}} f &= f\, p \end{aligned}$$

Left-to-right evaluation evidently produces unambiguous results. As an example, let us consider again the expression `x=0, x+(x=1)`, whose denotation is given below. The result of its evaluation is the value 1 and the final state is identical to the initial one, except that x has the value 1.

$$[\![\texttt{x=0, x+(x=1)}]\!] = \lambda s.\, \langle 1, s[\texttt{x} \mapsto 1] \rangle$$

3.2 Non-deterministic choice

We continue by considering evaluation strategies that allow for ambiguity in expression evaluation. In this case, it is necessary to replace P by a monad supporting multiple results. An obvious choice is the *powerdomain* monad, that is defined below using an appropriate powerdomain constructor such as Plotkin's convex powerdomain [Plot76]. Multiple results are constructed with the polymorphic operator $\cdot \uplus_{\mathsf{P}} \cdot : \mathsf{P}(A) \times \mathsf{P}(A) \to \mathsf{P}(A)$.

$$\begin{aligned} \mathsf{P}(T) &= T^\natural \\ \mathsf{unit}_{\mathsf{P}} &= \lambda v.\, \{\!| v |\!\} \\ p *_{\mathsf{P}} f &= \mathsf{ext}^\natural\, f\, p \\ p_1 \uplus_{\mathsf{P}} p_2 &= p_1 \uplus^\natural p_2 \end{aligned}$$

In the above $\{\!| \cdot |\!\} : A \to A^\natural$, $\mathsf{ext}^\natural : (A \to B^\natural) \to A^\natural \to B^\natural$ and $\cdot \uplus^\natural \cdot : A^\natural \times A^\natural \to A^\natural$ are standard operators of the convex powerdomain.

A simple evaluation strategy that allows for ambiguous results is the *non-deterministic choice*. According to this, operands may be evaluated in any order but the evaluation of each one is performed individually and no interleaving is possible. In the case of ELSE, non-deterministic choice specifies that the operands of + may be evaluated left-to-right or right-to-left but, in any case, evaluation of one of them will have been completed before evaluation of the other starts.

In order to formally specify non-deterministic choice in our semantic model, it suffices to redefine operator $\bowtie$ and include right-to-left evaluation as well. We do this by introducing a new polymorphic operator $\cdot \parallel \cdot : \mathsf{M}(A) \times \mathsf{M}(A) \to \mathsf{M}(A)$, which represents non-deterministic choice in computations and whose implementation depends on monad M. Given two computations $m_1 : \mathsf{M}(A)$ and $m_2 : \mathsf{M}(A)$, the set of possible results for computation $m_1 \parallel m_2 : \mathsf{M}(A)$ is the union of the sets of all possible results for both m_1 and m_2.[6]

$$\begin{aligned} & m_1 \parallel m_2 \; = \; \lambda s.\, m_1\, s \uplus_{\mathsf{P}} m_2\, s \\ & m_1 \bowtie m_2 \; = \\ & \quad m_1 *_{\mathsf{M}} (\lambda v_1.\, m_2 *_{\mathsf{M}} (\lambda v_2.\, \mathsf{unit}_{\mathsf{M}}\, \langle v_1, v_2 \rangle)) \parallel \\ & \qquad m_2 *_{\mathsf{M}} (\lambda v_2.\, m_1 *_{\mathsf{M}} (\lambda v_1.\, \mathsf{unit}_{\mathsf{M}}\, \langle v_1, v_2 \rangle)) \end{aligned}$$

It may be interesting to notice that, by defining $m_1 \parallel m_2 = m_1$, non-deterministic choice degenerates into left-to-right evaluation.

In this evaluation strategy, `x=0, x+(x=1)` may produce 1 or 2 and `x=0, x+(x=1, x=2, 0)` may produce 0 or 2, as shown below.

$$\begin{aligned} & [\![\texttt{x=0, x+(x=1)}]\!] \; = \\ & \quad \lambda s.\, \{\!| \langle 1, s[\texttt{x} \mapsto 1] \rangle, \langle 2, s[\texttt{x} \mapsto 1] \rangle |\!\} \\ & [\![\texttt{x=0, x+(x=1, x=2, 0)}]\!] \; = \\ & \quad \lambda s.\, \{\!| \langle 0, s[\texttt{x} \mapsto 2] \rangle, \langle 2, s[\texttt{x} \mapsto 2] \rangle |\!\} \end{aligned}$$

3.3 Interleaving

The notion of execution *interleaving* is a well known one in the theory of concurrency. An interleaved evaluation of an expression consists of an arbitrary merging of the *atomic steps* that constitute the evaluation of its subparts. In the case of ELSE it is natural to consider side effects, i.e. read and write accesses to the state, as the only kind of atomic steps. Furthermore, it is often useful to disable interleaving and evaluate an arbitrary expression in a single atomic step. To achieve this, we introduce a new construct in ELSE.

$$\begin{aligned} E : \mathbf{Expr} \; ::= \; & n \mid I \mid I\texttt{=}E \mid E_1\texttt{+}E_2 \mid \texttt{-}E \\ & \mid E_1\texttt{,}E_2 \mid \texttt{<}E\texttt{>} \end{aligned}$$

[5]Left-to-right evaluation is used by many programming languages, such as Standard ML and Java.

[6]Operator $\parallel$ would be useful on its own in a hypothetical extension of ELSE that would support non-determinism explicitly, e.g. in expressions of the form $E_1 ? E_2$.

Expression <E> is equivalent to E with the only difference that the former is evaluated in a single atomic step, and therefore no interleaving is permitted during its evaluation. The semantics of the newly introduced construct may be expressed using a new function exhaust : $\mathsf{M}(A) \to \mathsf{M}(A)$, that transforms a computation consisting of several steps to an equivalent one consisting of a single step.

$$[\![{<}E{>}]\!] \;=\; \mathsf{exhaust}\, [\![E]\!]$$

In the non-interleaving semantics of the previous sections, expressions <E> and E are equivalent and it suffices to take $\mathsf{exhaust} = \mathsf{id}$.

In order to formally specify interleaving in our semantics, it is necessary to modify the domain of denotations. We define monad M as follows, implementing thus a tree-like *branching semantics*. We retain the powerdomain monad P from the previous section.

$$\begin{aligned}
\mathsf{M}(T) &= T \oplus (\mathbf{S} \to \mathsf{P}(\mathsf{M}(T) \times \mathbf{S})) \\
\mathsf{unit}_\mathsf{M} &= \mathbf{inl} \\
m \ast_\mathsf{M} f &= \mathbf{fix}\,(\lambda\, g.\, [\, f, \lambda\, r.\, \mathbf{inr}\,(\lambda\, s.\, r\ s \ast_\mathsf{P} \\
&\qquad (\lambda\, \langle m', s' \rangle.\, \mathsf{unit}_\mathsf{P}\, \langle g\, m', s' \rangle))\,]\,)\; m \\
\mathsf{update} &= \lambda\, f.\, \mathbf{inr}\,(\lambda\, s.\, \mathsf{unit}_\mathsf{P}\, \langle \mathbf{inl}\, s, f\ s \rangle) \\
\mathsf{exhaust} &= \mathbf{fix}\,(\lambda\, g.\, [\, \mathbf{inl}, \lambda\, r.\, \mathbf{inr}\,(\lambda\, s.\, r\ s \ast_\mathsf{P} \\
&\qquad (\lambda\, \langle m', s' \rangle.\, [\, \lambda\, v'.\, \mathsf{unit}_\mathsf{P}\, \langle \mathbf{inl}\, v', s' \rangle, \\
&\qquad\quad \lambda\, r'.\, r'\ s'\,]\,(g\, m')))\,]\,)
\end{aligned}$$

In this semantics, the denotation of an expression is either a computed value or a function mapping the current state to a set of possible intermediate results, that may be produced after a single atomic step is performed. Each such result consists of the denotation of a partially computed expression and a new state. Such denotations are called *resumptions* and are frequently used in specifying the semantics of concurrency [Moss90, dBak96]. The resumption semantics that we use in this section is derived from the direct semantics of section 3.1.

We should emphasize here that expressions whose value is constant and does not depend on the state are considered as already computed and are denoted by elements of the left summand of $\mathsf{M}(T)$. In contrast to that, expressions whose computation requires one or more atomic steps to be performed and whose value depends on the state are denoted by elements of the right summand of $\mathsf{M}(T)$.

Operator $\|$ can be easily defined for the new semantics.[7] We notice again that by taking $m_1 \parallel m_2 = m_1$ interleaving degenerates into left-to-right evaluation.

$$\begin{aligned}
m_1 \parallel m_2 \;=\; &\mathbf{inr}\,(\lambda\, s.\, \mathbf{let}\ f = [\, \lambda\, v.\, \mathsf{unit}_\mathsf{P}\, \langle \mathbf{inl}\, v, s \rangle, \\
&\lambda\, r.\, r\ s\,]\ \mathbf{in}\ f\, m_1 \uplus_\mathsf{P} f\, m_2)
\end{aligned}$$

The definition of operator $\bowtie$ becomes more elegant by the introduction of the following two functions.

$$\begin{aligned}
\mathsf{analyze} &: \mathsf{M}(A) \to (A \to B) \times B \to B \\
\mathsf{analyze} &= \lambda\, m.\, \lambda\, \langle f, z \rangle.\, [\, f, \lambda\, r.\, z\,]\; m \\
\mathsf{apply} &: \mathsf{M}(A) \to (\mathsf{M}(A) \to \mathsf{M}(B)) \to \mathsf{M}(B) \\
\mathsf{apply} &= \lambda\, m.\, \lambda\, f.\, [\, f \circ \mathbf{inl}, \lambda\, r.\, \mathbf{inr}\,(\lambda\, s.\, r\ s \ast_\mathsf{P} \\
&\qquad (\lambda\, \langle m', s' \rangle.\, \mathsf{unit}_\mathsf{P}\, \langle f\ m', s' \rangle))\,]\; m
\end{aligned}$$

Function analyze can be used to distinguish between a computation that has been completed and one that requires atomic steps to be performed. Function apply performs the first atomic step of the computation specified by its first argument, if this is possible, and applies its second argument to the partial results. By using these two functions, operator $\bowtie$ can be defined as follows.

$$\begin{aligned}
m_1 \bowtie m_2 \;=\; &\mathbf{fix}\,(\lambda\, g.\, \lambda\, \langle m_1, m_2 \rangle. \\
&\mathsf{analyze}\, m_1\, \langle \lambda\, v_1.\, m_2 \ast_\mathsf{M} (\lambda\, v_2.\, \mathsf{unit}_\mathsf{M}\, \langle v_1, v_2 \rangle), \\
&\mathsf{analyze}\, m_2\, \langle \lambda\, v_2.\, m_1 \ast_\mathsf{M} (\lambda\, v_1.\, \mathsf{unit}_\mathsf{M}\, \langle v_1, v_2 \rangle), \\
&\mathsf{apply}\, m_1\, (\lambda\, m'_1.\, g\, \langle m'_1, m_2 \rangle) \parallel \\
&\quad \mathsf{apply}\, m_2\, (\lambda\, m'_2.\, g\, \langle m_1, m'_2 \rangle) \rangle \rangle)\; \langle m_1, m_2 \rangle
\end{aligned}$$

When both computations of m_1 and m_2 require atomic steps to be performed, operator $\bowtie$ must choose between performing an atomic step from m_1 or m_2.

Returning to our example `x=0, x+(x=1)`, resumption semantics produces 1 or 2 as possible results, exactly as non-deterministic choice. However, expression `x=0, x+(x=1, x=2, 0)` may now produce the result 1, in addition to the possible results 0 and 2 of non-deterministic choice, in case the (implicit) dereferencing of `x` is interleaved between atomic steps `x=1` and `x=2`. The denotations of both examples in equational and schematic form are shown in Figure 1. Their exhausted versions are also shown, so that the results of their (uninterrupted) evaluation can be examined.

3.4 Sequence points

A language that does not fully specify evaluation order and, at the same time, allows evaluation of expressions to produce side effects is an inherently ambiguous language.[8] However, for a programming language to be useful, ambiguities in program execution should be avoided as much as possible. In this section we focus on the ANSI C programming language which features the combination of characteristics mentioned before. In order to disallow undesired ambiguities, the C standard introduces restrictions imposed on expression evaluation. An attempt to define the semantics of these restrictions is made in this section.

The C standard is very careful not to overspecify, in order to allow for reasonable optimizations in implementations of the language. Such optimizations may see fit to postpone a side effect and, for instance, store a variable's value in a register instead of its appointed location in memory.

> At certain specified points in the execution sequence called *sequence points*, all side effects of previous evaluations shall be complete and

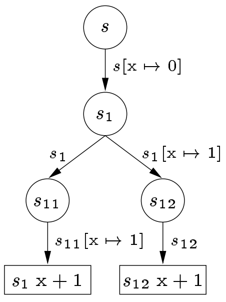
[7]However, our resumption semantics forces us to introduce an atomic step for the non-deterministic choice between two computed values. This could be avoided by using the more complicated monad $\mathsf{M}(T) = \mathsf{P}(T) \oplus (\mathbf{S} \to \mathsf{P}(\mathsf{M}(T) \times \mathbf{S}))$.

[8]Several programming languages, such as ALGOL 60, Pascal, C/C++ and Scheme, prefer not to impose a specific evaluation order on implementors.

Expression: `x=0, x+(x=1)`

$$
\begin{array}{l}
\llbracket \texttt{x=0, x+(x=1)} \rrbracket = \mathbf{inr}\;(\lambda\, s.\, \{\!\!\{ \\
\quad \langle \mathbf{inr}\;(\lambda\, s_1.\, \{\!\!\{ \\
\quad\quad \langle \mathbf{inr}\;(\lambda\, s_{11}.\, \{\!\!\{ \langle \mathbf{inl}\;(s_1\;\mathrm{x}+1), s_{11}[\mathrm{x} \mapsto 1] \rangle \}\!\!\}), s_1 \rangle, \\
\quad\quad \langle \mathbf{inr}\;(\lambda\, s_{12}.\, \{\!\!\{ \langle \mathbf{inl}\;(s_{12}\;\mathrm{x}+1), s_{12} \rangle \}\!\!\}), s_1[\mathrm{x} \mapsto 1] \rangle \\
\quad \}\!\!\}), s[\mathrm{x} \mapsto 0] \rangle \\
\}\!\!\})
\end{array}
$$

$$
\begin{array}{l}
\mathsf{exhaust}\; \llbracket \texttt{x=0, x+(x=1)} \rrbracket = \mathbf{inr}\;(\lambda\, s.\, \{\!\!\{ \\
\quad \langle \mathbf{inl}\;1, s[\mathrm{x} \mapsto 1] \rangle, \\
\quad \langle \mathbf{inl}\;2, s[\mathrm{x} \mapsto 1] \rangle \\
\}\!\!\})
\end{array}
$$

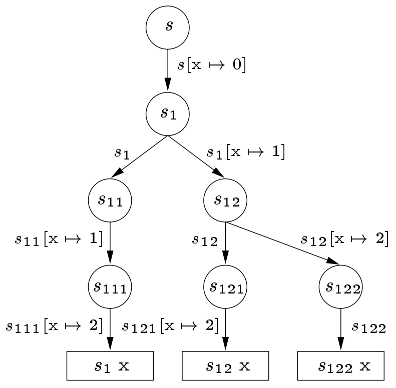


s
$s[\mathrm{x} \mapsto 0]$
s_1
s_1
$s_1[\mathrm{x} \mapsto 1]$
s_{11}
s_{12}
$s_{11}[\mathrm{x} \mapsto 1]$
s_{12}
$s_1\;\mathrm{x}+1$
$s_{12}\;\mathrm{x}+1$

Expression: `x=0, x+(x=1, x=2, 0)`

$$
\begin{array}{l}
\llbracket \texttt{x=0, x+(x=1, x=2, 0)} \rrbracket = \mathbf{inr}\;(\lambda\, s.\, \{\!\!\{ \\
\quad \langle \mathbf{inr}\;(\lambda\, s_1.\, \{\!\!\{ \\
\quad\quad \langle \mathbf{inr}\;(\lambda\, s_{11}.\, \{\!\!\{ \\
\quad\quad\quad \langle \mathbf{inr}\;(\lambda\, s_{111}.\, \{\!\!\{ \langle \mathbf{inl}\;(s_1\;\mathrm{x}), s_{111}[\mathrm{x} \mapsto 2] \rangle \}\!\!\}), s_{11}[\mathrm{x} \mapsto 1] \rangle \\
\quad\quad \}\!\!\}), s_1 \rangle, \\
\quad\quad \langle \mathbf{inr}\;(\lambda\, s_{12}.\, \{\!\!\{ \\
\quad\quad\quad \langle \mathbf{inr}\;(\lambda\, s_{121}.\, \{\!\!\{ \langle \mathbf{inl}\;(s_{12}\;\mathrm{x}), s_{121}[\mathrm{x} \mapsto 2] \rangle \}\!\!\}), s_{12} \rangle, \\
\quad\quad\quad \langle \mathbf{inr}\;(\lambda\, s_{122}.\, \{\!\!\{ \langle \mathbf{inl}\;(s_{122}\;\mathrm{x}), s_{122} \rangle \}\!\!\}), s_{12}[\mathrm{x} \mapsto 2] \rangle \\
\quad\quad \}\!\!\}), s_1[\mathrm{x} \mapsto 1] \rangle \\
\quad \}\!\!\}), s[\mathrm{x} \mapsto 0] \rangle \\
\}\!\!\})
\end{array}
$$

$$
\begin{array}{l}
\mathsf{exhaust}\; \llbracket \texttt{x=0, x+(x=1, x=2, 0)} \rrbracket = \mathbf{inr}\;(\lambda\, s.\, \{\!\!\{ \\
\quad \langle \mathbf{inl}\;0, s[\mathrm{x} \mapsto 2] \rangle, \\
\quad \langle \mathbf{inl}\;1, s[\mathrm{x} \mapsto 2] \rangle, \\
\quad \langle \mathbf{inl}\;2, s[\mathrm{x} \mapsto 2] \rangle \\
\}\!\!\})
\end{array}
$$



s
$s[\mathrm{x} \mapsto 0]$
s_1
s_1
$s_1[\mathrm{x} \mapsto 1]$
s_{11}
s_{12}
$s_{11}[\mathrm{x} \mapsto 1]$
s_{12}
$s_{12}[\mathrm{x} \mapsto 2]$
s_{111}
s_{121}
s_{122}
$s_{111}[\mathrm{x} \mapsto 2]$
$s_{121}[\mathrm{x} \mapsto 2]$
s_{122}
$s_1\;\mathrm{x}$
$s_{12}\;\mathrm{x}$
$s_{122}\;\mathrm{x}$

Figure 1: Two examples of interleaving semantics.

> no side effects of subsequent evaluations shall have taken place. [ANSI90, §5.1.2.3]

Sequence points are the standard's guarantee that all side effects will eventually take place properly and that optimizations will not affect normal execution. Additional restrictions are imposed to disallow excessively ambiguous expressions.

> Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be accessed only to determine the value to be stored. [ANSI90, §6.3]

The first restriction disallows expressions such as `(x=1)+(x=2)`. The second is careful enough to disallow `x+(x=1)` but not `x=x+1`. We should notice however that these restrictions do not completely eliminate ambiguity in C expressions. Function calls are always surrounded by sequence points and do not allow interleaving in their execution. Therefore, an expression such as `f()+g()` does not violate the sequence point restrictions and is ambiguous when both `f()` and `g()` produce side effects, since the order in which they will be called is not specified by the standard.

In order to study the semantics of ANSI C's mechanism of sequence points, we modify our example language. We first change the semantics of the comma operator, by enforcing a sequence point between the (left-to-right) evaluation of the two operands. We also add a unary operator `#`, which enforces two sequence points: one just before and one just after the evaluation of its operand. Although ELSE still does not feature functions, an expression of the form `<#E>` is a reasonably equivalent to a call to a C function with no arguments.

$$
\begin{array}{rcl}
E : \textbf{Expr} & ::= & n \mid I \mid I\texttt{=}E \mid E_1\texttt{+}E_2 \mid \texttt{-}E \\
& \mid & E_1\texttt{,}E_2 \mid \texttt{<}E\texttt{>} \mid \texttt{\#}E
\end{array}
$$

The required changes in the semantics of ELSE are given by the following equations, where $\mathsf{seqpt} : \mathsf{M}(\mathbf{U})$ represents a computation that enforces a sequence point.

$$
\begin{array}{rcl}
\llbracket E_1\texttt{,}E_2 \rrbracket & = & \llbracket E_1 \rrbracket \;*_{\mathsf{M}}\; (\lambda\, n.\, \mathsf{seqpt} \;*_{\mathsf{M}}\; (\lambda\, u.\, \llbracket E_2 \rrbracket)) \\
\llbracket \texttt{\#}E \rrbracket & = & \mathsf{seqpt} \;*_{\mathsf{M}}\; (\lambda\, u_1.\, \llbracket E \rrbracket \;*_{\mathsf{M}} \\
& & \quad (\lambda\, n.\, \mathsf{seqpt} \;*_{\mathsf{M}}\; (\lambda\, u_2.\, \mathsf{unit}_{\mathsf{M}}\; n)))
\end{array}
$$

By taking $\mathsf{seqpt} = \mathsf{unit}_{\mathsf{M}}\; \mathbf{u}$, the semantics of this section degenerates into that of the previous section.

To distinguish between a side effect that has taken place and one that has not, it is necessary to change the definition of state. By taking into account the additional restrictions imposed by the mechanism of sequence points, it is possible to define $\mathbf{S}$ as a pair of two functions, the first representing the memory and the second representing the pending side effects. Knowing that at most one writing to each memory location takes place between two successive sequence points and that no reading takes place after the writing, we may specify that all side effects actually take place at the following sequence point by a call to function $\mathsf{commit} : \mathbf{S} \to \mathbf{S}$. The new definition of $\mathbf{S}$ is given below.

$$\begin{array}{lcl}
\mathbf{S} & = & (\mathbf{Ide} \to \mathbf{N} \oplus \mathbf{U}) \times (\mathbf{Ide} \to \mathbf{N} \oplus \mathbf{U}) \\
s_\circ & = & \langle \lambda\, I.\, \mathbf{inr}\,\mathbf{u}, \lambda\, I.\, \mathbf{inl}\,\mathbf{u} \rangle \\
\mathsf{get} & = & \lambda\, I.\, \lambda\, \langle s_n, s_x \rangle. \\
 & & \quad [\, \bot, \lambda\, u.\, [\, \mathsf{id}, \bot \,]\, (s_n\; I) \,]\, (s_x\; I) \\
\mathsf{put} & = & \lambda\, I.\, \lambda\, n.\, \lambda\, s. \\
 & & \quad [\, \bot, \lambda\, u.\, \langle s_n, s_x[I \mapsto \mathbf{inl}\, n] \rangle \,]\, (s_x\; I) \\
\mathsf{commit} & = & \lambda\, \langle s_n, s_x \rangle. \\
 & & \quad \mathbf{let}\ s'_n = \lambda\, I.\, [\, \mathbf{inl}, \lambda\, u.\; s_n\; I \,]\, (s_x\; I) \\
 & & \quad \mathbf{in}\ \ \langle s'_n, \lambda\, I.\, \mathbf{inr}\,\mathbf{u} \rangle
\end{array}$$

Implementations of get and put guarantee that no reading or writing is permitted if a side effect for the same variable is pending. Subsequently, we can define seqpt as follows.

$$\mathsf{seqpt} \;=\; \mathsf{update}\;\mathsf{commit}\; *_{\mathsf{M}}\; (\lambda\, s.\, \mathsf{unit}_{\mathsf{M}}\, \mathbf{u})$$

By looking at the definition of seqpt, it is easy to see that the proposed semantics satisfies the requirement that all side effects shall be complete at sequence points. The definition of put also satisfies the requirement that an object shall have its value modified at most once between successive sequence points. Finally, the requirement that the prior value shall be accessed only to determine the new value is indirectly satisfied by this semantics:

- A read access that determines the new value will certainly precede the write access in all possible evaluation orders. It is therefore always allowed by the definition of get.

- A read access that does not determine the new value is only allowed by the definition of get if it precedes the write access in the evaluation order. But it can be easily proved that there exists an evaluation order in which this is not true and, for this particular evaluation order, the definition of get produces an error. This error will propagate in the denotation of the expression and, therefore, if we rule out expressions containing errors as possible results, our semantics satisfies the requirement.

Of course, it is possible to define a semantics satisfying this last requirement directly, by defining **S** in such a way as to keep track of performed read accesses.

As an example, we consider again the expression `x=0, x+(x=1)`. The semantics shown in Figure 2 is similar to the one in Figure 1 with the exception of the extra commit step.[9] In the exhausted version, we can see that the result 2 has been replaced by an error, since read access s_{112} x is performed on state $s_{11}[\mathtt{x} \mapsto 1]$, without an intermediate sequence point.

4 Related work

Research in the field of expression languages, side effects and evaluation order spans a wide area of interest. Among the earliest related publications we should mention the work of Boehm [Boeh82] and Kowaltowski [Kowa77] on the axiomatic semantics of expression languages with side effects. Both avoid the pitfall of evaluation order by assuming left-to-right evaluation. The work of Filinski [Fili96] focuses on the introduction of various kinds of effects in functional languages, using appropriate monads for state and continuations, but does not address the issue of evaluation order semantics. Evaluation order analysis in lazy functional languages is dealt with in the work of Draghicescu and Purushothaman [Drag90] and Bloss [Blos94]. Both do not define a semantics of execution for different evaluation strategies and are primarily interested in optimizations and the destructive update problem.

The semantics of many popular programming languages has been formally specified in literature, at least partially. Unspecified evaluation order is modeled mostly using permutation oracles or appropriate transition systems. Often the issue is excluded from the formal description and a convenient assumption is made. To the best of our knowledge, techniques as the one used in this paper have not been used for this purpose in relation with "real-world" programming languages.

Significant research has been conducted recently concerning semantic aspects of the C programming language, mainly because of the language's popularity and its wide applications. In what seems to be the earliest formal approach, Sethi addresses the semantics of pre-ANSI C, using the denotational approach and assuming left-to-right evaluation of expressions [Seth80]. In the work of Gurevich and Huggins [Gure93] a formal semantics for C is given in the form of evolving algebra. The semantics of evaluation order is based on the assumptions that no interleaving is possible in expression evaluation (i.e. non-deterministic choice as described in Section 3.2 is used) and that side effects take place as they are generated. Both assumptions do not agree with the ANSI standard. A higher-level axiomatic semantics is proposed by Black and Windley [Blac96], which removes side effects from expressions and treats them as separate statements. In the work of Cook and Subramanian [Cook94b] a semantics for C is developed in the theorem prover Nqthm. It employs an oracle for determining evaluation order and order of side-effects, but this is not really used since the authors consider a subset of C with pure expressions. Cook et al. have also developed a denotational semantics for C based on temporal logic [Cook94a]. Although left-to-right evaluation is assumed in this work, the authors suggest how this can be remedied. However, it is not clear whether the suggestion allows for interleaving and there is no treatment of sequence points. To the best of our knowledge, the only semantics of ANSI C that correctly models unspecified order of evaluation, side effects and sequence points is defined in the work of Norrish [Norr97] in the form of operational semantics with small-step reductions. No similar denotational approach is known to us.

5 Conclusion and future work

This paper was mainly concerned with exploring the semantics of evaluation order in strict expression languages with side effects. The semantics of the example language ELSE was developed under various different evaluation strategies in a unified way, based on monad notation which

[9] To reduce complexity in the figure, we keep the old notation instead of the newly defined state operations get and put.

Expression: `x=0, x+(x=1)`

$$
\begin{array}{l}
[\![\texttt{x=0, x+(x=1)}]\!] = \textbf{inr}\ (\lambda\, s.\ \{\!\!\{ \\
\quad \langle \textbf{inr}\ (\lambda\, s_1.\ \{\!\!\{ \\
\quad\quad \langle \textbf{inr}\ (\lambda\, s_{11}.\ \{\!\!\{ \\
\quad\quad\quad \langle \textbf{inr}\ (\lambda\, s_{111}.\ \{\!\!\{ \langle \textbf{inl}\ (s_{11}\ \mathsf{x}+1), s_{111}[\mathsf{x} \mapsto 1] \rangle \}\!\!\}), s_{11} \rangle, \\
\quad\quad\quad \langle \textbf{inr}\ (\lambda\, s_{112}.\ \{\!\!\{ \langle \textbf{inl}\ (s_{112}\ \mathsf{x}+1), s_{112} \rangle \}\!\!\}), s_{11}[\mathsf{x} \mapsto 1] \rangle \\
\quad\quad \}\!\!\}), \mathsf{commit}\ s_1 \rangle \\
\quad \}\!\!\}), s[\mathsf{x} \mapsto 0] \rangle \\
\}\!\!\})
\end{array}
$$

$$
\begin{array}{l}
\mathsf{exhaust}\ [\![\texttt{x=0, x+(x=1)}]\!] = \textbf{inr}\ (\lambda\, s.\ \{\!\!\{ \\
\quad \langle \textbf{inl}\ 1, s[\mathsf{x} \mapsto 1] \rangle, \\
\quad \langle \bot, s[\mathsf{x} \mapsto 1] \rangle \\
\}\!\!\})
\end{array}
$$

$$\begin{tikzcd}
& s \arrow[d, "{s[\mathsf{x}\mapsto 0]}"] & \\
& s_1 \arrow[d, "\mathsf{commit}\ s_1"] & \\
& s_{11} \arrow[dl, "s_{11}"'] \arrow[dr, "{s_{11}[\mathsf{x}\mapsto 1]}"] & \\
s_{111} \arrow[d, "{s_{111}[\mathsf{x}\mapsto 1]}"] & & s_{112} \arrow[d, "s_{112}"] \\
\boxed{s_{11}\ \mathsf{x}+1} & & \boxed{s_{112}\ \mathsf{x}+1}
\end{tikzcd}$$

Figure 2: Interleaving semantics with sequence points.

improves the modularity and elegance of the result. The final version of ELSE is a significant subset of the language of C expressions and its semantics has been successfully modelled, including unspecified evaluation order, order of side effects and the mechanism of sequence points. This is a valuable result that opens the road towards a complete denotational description of ANSI C. Although ELSE is not capable of non-termination, the semantics is powerful enough to model it. Variable aliasing can be included by separating environment and state and the direct semantics in the definition of monad M can be easily replaced by continuations, which will be needed in a complete formal treatment of C. It is also easy to extend operator $\bowtie$ to for any number of operands.

Future work will have to proceed on several fronts. On the one hand it would be interesting to relate the denotational semantics described in this paper to operational semantics for the same languages, by equations of the form $\mathcal{O} = \mathsf{abs}\ \mathcal{D}$, and study the issue of full abstraction in the style of [dBak96]. This will be rather complicated, primarily because of monad notation, but we believe that it is possible although we still lack a firm proof. On the other hand, the replacement of domains with functor categories in the foundations of our model should be considered. In this way, a possible worlds semantics of evaluation order will be feasible, which can subsequently be used as a basis for the abstract treatment of local variables. Finally, this work is part of an ongoing bigger project, aiming at the development of a denotational semantics for the complete ANSI C language, in which the resumption semantics of Section 3.4 will have a prominent role.

References

[ANSI90] American National Standards Institute, New York, NY, *ANSI/ISO 9899-1990, American National Standard for Programming Languages: C*, 1990, Revision and redesignation of ANSI X3.159-1989.

[Blac96] P. E. Black and P. J. Windley, "Inference Rules for Programming Languages with Side Effects in Expressions", in *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'96)*, pp. 51–60, Turku, Finland, 26–30 August 1996, Springer Verlag.

[Blos94] A. Bloss, "Path Analysis and the Optimization of Nonstrict Functional Languages", *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, pp. 328–369, May 1994.

[Boeh82] H. J. Boehm, "A Logic for Expressions with Side Effects", in *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 268–280, 1982.

[Cook94a] J. Cook, E. Cohen and T. Redmond, "A Formal Denotational Semantics for C", Technical Report 409D, Trusted Information Systems, September 1994.

[Cook94b] J. Cook and S. Subramanian, "A Formal Semantics for C in Nqthm", Technical Report 517D, Trusted Information Systems, October 1994.

[dBak96] J. de Bakker and E. de Vink, *Control Flow Semantics*, Foundations of Computing Series, MIT Press, Cambridge, MA, 1996.

[Dijk76] E. W. Dijkstra, *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, NJ, 1976.

[Drag90] M. Draghicescu and S. Purushothaman, "A Compositional Analysis of Evaluation-Order and its Application", in *Conference Record of the ACM Symposium on Lisp and Functional Programming*, pp. 242–250, Nice, France, 1990.

[Fili96] A. Filinski, *Controlling Effects*, Ph.D. thesis, Carnegie Mellon University, School of Computer Science, May 1996, Also as Technical Report CMU-CS-96-119.

[Gure93] Y. Gurevich and J. K. Huggins, "The Semantics of the C Programming Language", in E. Börger et al., editors, *Selected Papers from CSL'92 (Computer Science Logic)*, vol. 702 of *Lecture Notes in Computer Science*, pp. 274–308, Springer Verlag, New York, NY, 1993.

[Henn90] M. Hennessy, *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*, John Wiley and Sons, New York, NY, 1990.

[Kowa77] T. Kowaltowski, "Axiomatic Approach to Side Effects and General Jumps", *Acta Informatica*, vol. 7, pp. 357–360, 1977.

[Lian95] S. Liang, P. Hudak and M. Jones, "Monad Transformers and Modular Interpreters", in *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, San Francisco, CA, January 1995.

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

[Moss90] P. D. Mosses, "Denotational Semantics", in J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, vol. B, chapter 11, pp. 577–631, Elsevier Science Publishers B.V., 1990.

[Norr97] M. Norrish, "An Abstract Dynamic Semantics for C", Technical Report TR-421, University of Cambridge, Computer Laboratory, May 1997.

[Plot76] G. D. Plotkin, "A Powerdomain Construction", *SIAM Journal on Computing*, vol. 5, pp. 452–487, 1976.

[Seth80] R. Sethi, "A Case Study in Specifying the Semantics of a Programming Language", in *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages*, pp. 117–130, January 1980.

[Stoy77] J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, MA, 1977.

[Wadl92] P. Wadler, "The Essence of Functional Programming", in *Proceedings of the 19th Annual Symposium on Principles of Programming Languages (POPL'92)*, January 1992.