J. Functional Programming **10** (3): 227–244, May 2000. Printed in the United Kingdom

© 2000 Cambridge University Press

A study of evaluation order semantics in expressions with side effects

NIKOLAOS S. PAPASPYROU

Department of Electrical and Computer Engineering, Division of Computer Science, Software Engineering Laboratory, National Technical University of Athens, Polytechnioupoli, 15780 Zografou, Athens, Greece
(*e-mail:* nickie@softlab.ntua.gr)

DRAGAN MAĆOŠ

debis Systemhaus GEI, debis Haus am Potsdamer Platz, Eichhornstrasse 3, 10875 Berlin, Germany
(*e-mail:* dmacos@debis.com)

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 specific order is enforced, the language becomes non-deterministic. In this paper we study the denotational 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 a dialect of Haskell as a metalanguage, instead of mathematical notation, and use monads and monad transformers to improve modularity. In this way, only small modifications are required for each transition. The result is a better understanding of different evaluation strategies and a unified way of specifying their semantics. Furthermore, a significant step is achieved towards a complete and accurate semantics for ANSI C.

Capsule Review

This work demonstrates a neat application of monad transformers: the study of evaluation-order semantics in a programming language with side effects. The key insight is to abstract away the specification of the evaluation-order mechanism in the denotational semantics of an imperative language, in such a way that it can be instantiated easily with different strategies. This results in a highly-modular, easy-to-understand, semantics. If you start with a monadic denotational semantics, the most natural way to realize the abstraction is the use of monad transformers (and one might speculate that a similar approach using continuation semantics would result in the use of continuation transformers). This work is not just an exercise: the authors show that it can be a helpful tool in the understanding of at least one important language: ANSI C.

https://doi.org/10.1017/S095679680000366X Published online by Cambridge University Press

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 upon, but may not alter, the program state. For such 'ideal' languages several theories exist that can be used to prove the properties of programs, starting from Hoare's work on axiomatic semantics in the late 1960s. 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, since different evaluation strategies may lead to different results.

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 eager and impure expression language is studied under various different 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 accurately and 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 (Papaspyrou, 1998).

The formal semantics of pure expressions can usually be easily specified in operational, denotational or axiomatic form. 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, monads and monad transformers has been proposed as a remedy, and has become quite popular both in the denotational semantics and the functional programming community. Comprehensive introductions can be found in the work of Moggi (1990), Wadler (1992) and Liang, Hudak and Jones (1995). Monads and monad transformers are used in this paper, and it is demonstrated that, as a result, the semantics is significantly improved in terms of modularity and elegance.

https://doi.org/10.1017/S095679680000366X Published online by Cambridge University Press

Instead of using mathematical notation, a functional programming language is used for the definition of the denotational semantics. This language is the dialect of *Haskell* (Peyton Jones and Hughes, 1999) implemented by Mark Jones' Hugs 1.3c,[1] supporting multiple parameter type classes[2] (Peyton Jones *et al.*, 1997). For simplicity, this dialect of Haskell is referred to in the following sections of this paper as *Haskell*.[3] It has been chosen over ordinary Haskell, since the latter's higher-order type classes are not adequate to formulate monads and monad transformers in a direct and elegant way.[4] Moreover, by using a programming language instead of mathematical notation, we allow our denotational semantics to be directly implementable, in the form of an interpreter for the languages under study, and thus to be easily tested and evaluated.

The structure of this paper is as follows. Section 2 defines the example language and briefly discusses semantic issues related to evaluation order. Section 3 contains the necessary background definitions of monads, states and monad transformers. In section 4 a denotational semantics is gradually built for several different evaluation strategies. Section 5 shows how our work relates to the literature, and section 6 concludes with some final remarks.

2 The example language ELSE

Consider a simple eager 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 4.4 and 4.5.

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

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). More complex language constructs are not included in ELSE, since they are of no help in our study of evaluation order and side effects. The informal semantics of ELSE expressions is certainly familiar to most readers. The value of $I{=}E$ is the value of E and the value of $E_1{,}E_2$ is the value of E_2. Furthermore, evaluation of $I{=}E$ generates a side-effect by assigning the value of E to variable I. For simplicity, ELSE supports a single global scope of integer variables, whose values are initially undefined. Variables need not be declared.

The same operator precedence and associativity as in ANSI C is adopted for ELSE. Parentheses are used to group expressions. All these details are hidden in the abstract syntax. Apart from the comma operator, which always evaluates its left operand

[1] Hugs 1.3c can be obtained from `ftp://ftp.cs.nott.ac.uk/haskell/hugs/hugs13/`.

[2] Multiple parameter type classes have also been implemented in the GHC 3.02 (and later) Haskell compiler, as well as in Hugs 1.4 and the native mode of Hugs 98.

[3] A large number of Haskell extensions, including multiple parameter type classes, are currently being considered favourably for the evolving standard of Haskell 2. See `http://haskell.org/` for details. This fact justifies the simplified name for the dialect that is used here.

[4] As an alternative, the language *Gofer* (Jones, 1994) could be used with minor changes in the source code that is given in the present paper. Gofer is also based on a subset of Haskell extended with constructor classes (Jones, 1995).

https://doi.org/10.1017/S095679680000366X Published online by Cambridge University Press

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. In fact, different evaluation orders are only possible in the presence of operator $+$, since this is the only binary operator in ELSE with unspecified evaluation order. It should be clear, however, that this lack of specification renders the evaluation of ELSE expressions ambiguous, as different evaluation strategies may lead to different results. As an example, consider the simple expression $x = 0, x + (x = 1)$ whose evaluation may result in 1 or 2, depending on whether the (implicit) dereferencing of x in the left operand of $+$ will take place before or after the assignment in the right operand.

The following code implements the abstract syntax of ELSE in Haskell. Types `Ide` and `N` represent identifiers and integer values, whereas type `Expr` represents ELSE expressions.

```
type Ide = String
type N   = Int

data Expr = E_int N | E_ide Ide | E_assign (Ide, Expr) | E_plus (Expr, Expr)
          | E_neg Expr | E_comma (Expr, Expr)
```

Also, the following code implements two examples. The first corresponds to the simple expression $x = 0, x + (x = 1)$ that was discussed before. The second corresponds to the slightly more complex expression $x = 0, x + (x = 1, x = 2, 0)$, whose result can be 0, 1 or 2 under different evaluation strategies. Again, what determines the final result is the order in which the (implicit) dereferencing of x will take place.

```
ex1 = E_comma (E_assign ("x", E_int 0),
               E_plus (E_ide "x", E_assign ("x", E_int 1)))

ex2 = E_comma (E_assign ("x", E_int 0),
               E_plus (E_ide "x",
                       E_comma (E_assign ("x", E_int 1),
                       E_comma (E_assign ("x", E_int 2),
                                E_int 0))))
```

3 Monads, states and monad transformers

The semantics of ELSE is defined in an abstract form using monads to improve modularity and elegance of notation. In brief, a monad is a triple of the form (`M`, `return`, `>>=`). `M` is a type constructor of kind $* \to *$, `return :: a -> M a` and `(>>=) :: M a -> (a -> M b) -> M b` are polymorphic functions for arbitrary types `a` and `b`, satisfying three monad laws.[5] Function `>>=` is used as an infix binary operator.

Types constructed by monad `M` denote *computations*, e.g. the type `M a` denotes computations returning values of type `a`. The definition of `M` reflects 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 `return v` is simply

[5] This description, although naïve, is sufficient for the purpose of this paper. Although proofs are omitted, all monads and monad transformers defined here satisfy the required properties (subject to some termination conditions) as shown in the first author's thesis (Papaspyrou, 1998).

https://doi.org/10.1017/S095679680000366X Published online by Cambridge University Press

a computation returning the value `v` and the result of `m >>= f` is the combined computation of `m`, returning `v`, followed by computation `f v`. Evaluation order is thus implicitly specified when using `>>=`, since the result of the first computation is needed before the second computation can begin. By using different monads, it is possible to derive different flavours of the semantics of ELSE without having to change the semantic equations each time.

The following code contains the definition of the standard Haskell class `Monad`, based on the previous description. The first line defines `>>=` as an infix left associative operator of very low precedence.

```
infixl 1 >>=

class Monad m where
   return :: a -> m a
   (>>=)  :: m a -> (a -> m b) -> m b
```

It is also useful to distinguish two subclasses of monads with additional features. Class `MultiMonad` represents monads with a binary operation `+|+` (*multiplication*) upon their elements, as shown below, and class `StrongMonad` represents monads with a binary operation `+:+` (*tensorial strength*). Although intuitively these two operations are used in order to express disjunction and conjunction respectively, with `+|+` indicating an option between two alternative computations and `+:+` indicating a combination of two simultaneous computations, their exact behaviour depends on a monad's definition.

```
class Monad m => MultiMonad m where
   (+|+) :: m a -> m a -> m a

class Monad m => StrongMonad m where
   (+:+) :: m a -> m b -> m (a, b)
```

The notion of *state* is a very important one in the study of the impure language ELSE. A state is an element of a type which supports two main operations, `load` and `store`, for retrieving and updating the contents of a variable in memory. A distinguished element of this type is the `initial` state, typically a state with all variables uninitialized. The class `State` is the common denominator of all possible state types, according to this description.

```
class State s where
   initial :: s
   load    :: Ide -> s -> N
   store   :: Ide -> N -> s -> s
```

A class of monads that are aware of the state is also useful. Class `StateMonad` supports two operations as an interface between computations and the state, in the style used by Liang, Hudak and Jones (1995). Function `setState` updates the state by applying its argument and returns a computation of the old state. Function `getState` simply returns a computation of the current state and can be implemented in terms of `setState`, as shown below. Parameter `s` represents the type of the state.

```
class Monad m => StateMonad s m where
   setState :: (s -> s) -> m s
   getState :: m s
   getState = setState id
```

https://doi.org/10.1017/S095679680000366X Published online by Cambridge University Press

Monad transformers are mappings between monads.[6] They are implemented in Haskell as higher-order type constructors of kind $(* \to *) \to * \to *$. The intuition behind them is that, if `T` is a monad transformer and `M` is a monad, then `T M` is also a monad and its properties are defined in terms of the properties of `M`. No special class of monad transformers needs be defined for the purpose of this paper.

4 Semantics of ELSE

The semantics of ELSE is defined as a function `sem :: Expr -> M N`, mapping expressions to integer computations. Monad `M` needs to be appropriately defined, in order to specify the characteristics of computations. Obviously, `M` must be an instance of class `StateMonad` for some appropriate state type `S`, to support state operations. Less obviously, it must be an instance of class `StrongMonad`, which is the mechanism by which evaluation order can be abstracted out of the semantics. In particular, operation `+:+` of `M` is used in the semantics of the binary operator `+`, which is the only construct of ELSE where evaluation order is an issue.

The semantic equations for all ELSE constructs are given below.[7]

```
sem :: Expr -> M N
sem (E_int n) =
   return n
sem (E_ide i) =
   getState >>= \(s :: S) -> return (load i s)
sem (E_assign (i, e)) =
   sem e >>= \n -> setState (store i n) >>= \(s :: S) -> return n
sem (E_plus (e1, e2)) =
   sem e1 +:+ sem e2 >>= \(n1, n2) -> return (n1 + n2)
sem (E_neg e) =
   sem e >>= \n -> return (-n)
sem (E_comma (e1, e2)) =
   sem e1 >>= \n -> sem e2
```

The implementations of `M` and `S` determine the intended semantics of ELSE, as far as evaluation order and side effects are concerned. The following sections discuss several implementations reflecting various possible evaluation strategies.

4.1 Left-to-right evaluation

A very simple, common and natural evaluation order is *left-to-right*.[8] In the case of ELSE, left-to-right evaluation specifies that the left operand of $+$ is evaluated completely before the right operand. A simple state type `S` can be defined as follows. The same state type will be used until Section 4.5.[9]

[6] Other definitions impose additional operations and restrictions on monad transformers, e.g. in the work of Espinosa (1995) and Liang (1998). Since these are not required for the purpose of this paper, the definition is simplified.

[7] Notice that the type of `s` in the second and third equations must be explicitly specified, since Haskell's type system can only infer that the type of `s` is an instance of class `State`, but not the exact type.

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

[9] The definition of function `update` can be found in the appendix.

https://doi.org/10.1017/S095679680000366X Published online by Cambridge University Press

```
newtype S = S (Ide -> N)

instance State S where
   initial = S (\i -> error ("Variable " ++ i ++ " has not been initialized"))
   load i (S s) = s i
   store i n (S s) = S (update i n s)
```

Before we define monad `M`, it is useful to define a monad transformer implementing the *direct semantics* approach, as a provision for the following sections. For every state type `s` given as a parameter, a monad transformer `D s` can be defined as follows. Parameter `m` specifies the monad representing the stateless computations.

```
newtype D s m a = D (s -> m (a, s))

instance Monad m => Monad (D s m) where
   return v   = D (\s -> return (v, s))
   D r >>= f  = D (\s -> r s >>= \(v', s') -> let D r' = f v' in r' s')
```

Monads constructed using `D` are aware of the state and therefore monad `D s m` is an instance of `StateMonad` for state type `s`. The following code implements this property:

```
instance Monad m => StateMonad s (D s m) where
   setState f = D (\s -> return (s, f s))
```

For the purpose of this section, the *identity monad* `Id` is a reasonable choice for stateless computations, ending up with the conventional direct semantics monad `M`.

```
newtype Id a = Id a

instance Monad Id where
   return x    = Id x
   Id x >>= f  = f x

type M a = D S Id a
```

To complete the semantics, we need to define `M` as an instance of `StrongMonad`. According to the left-to-right evaluation strategy, the following property can be specified for the direct semantics monad transformer. Order of evaluation is enforced by means of operator `>>=`.

```
instance Monad m => StrongMonad (D s m) where
   d1 +:+ d2 = d1 >>= \v1 -> d2 >>= \v2 -> return (v1, v2)
```

Left-to-right evaluation produces unambiguous results. To illustrate our semantics, let us consider the two examples of section 2. Expression $x = 0, x+(x = 1)$ evaluates to 1 and expression $x = 0, x + (x = 1, x = 2, 0)$ evaluates to 0. In both cases, the final state is identical to the initial one, except for the value of x which becomes 1 and 2, respectively. The following output is produced by the Haskell environment when evaluating the two examples.[10]

```
? sem ex1
(1,state)
? sem ex2
(0,state)
```

[10] As shown in the appendix, method `show` for the printing of computations starts evaluating from the `initial` state. The same method for states only prints the string ‘state’ instead of a state’s contents.

https://doi.org/10.1017/S095679680000366X Published online by Cambridge University Press

4.2 Non-deterministic choice

We continue by considering evaluation strategies that allow for ambiguity in expression evaluation. In this case, it is necessary to replace the identity monad `Id` for stateless computations by a monad supporting multiple results. Class `MultiMonad` is the mechanism by which this can be abstracted out of the semantics. In particular, operation `+|+` is a generic 'union' operation for computation results. The standard *list monad*,[11] denoted in Haskell by `[]`, is an obvious choice as a replacement for `Id`. It can be defined as an instance of `MultiMonad` by taking operator `+|+` as a synonym for list concatenation.

```
instance MultiMonad [] where
   (+|+) = (++)
```

Before we can use the direct semantics monad transformer to introduce states in computations with multiple results, it is necessary to define an additional property of `D`. If `m` is an instance of `MultiMonad`, then `D s m` is also an instance of `MultiMonad`. The two alternative computations start at the same initial state and their results are combined by using the union operator.

```
instance MultiMonad m => MultiMonad (D s m) where
   D r1 +|+ D r2 = D (\s -> r1 s +|+ r2 s)
```

With the list monad as the basis, we can easily modify the definition of `M` to allow for multiple results, as shown below.

```
type M a = D S [] a
```

A simple evaluation strategy which 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, the evaluation of one of them will be complete before the evaluation of the other starts.

To implement non-deterministic choice, it suffices to redefine how monad `M` is an instance of class `StrongMonad`. Again, a general property of the direct semantics monad transformer is given, on condition that the stateless monad supports multiple results. This time, two alternative computations are possible. The first operand of `+|+` represents left-to-right evaluation, while the second represents right-to-left evaluation.

```
instance MultiMonad m => StrongMonad (D s m) where
   d1 +:+ d2 =
      (d1 >>= \v1 -> d2 >>= \v2 -> return (v1, v2)) +|+
      (d2 >>= \v2 -> d1 >>= \v1 -> return (v1, v2))
```

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

```
? sem ex1
[(1,state),(2,state)]
? sem ex2
[(0,state),(2,state)]
```

[11] The list monad is predefined in Haskell as shown in the appendix.

https://doi.org/10.1017/S095679680000366X Published online by Cambridge University Press

4.3 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.

To implement interleaving in our semantics, it is necessary to modify the computation monad `M`. We define a *resumption monad transformer* `R` which implements a tree-like *branching semantics*. Resumptions are constructs which split a computation in a single atomic step (to be executed first) and a *resumed part*, which corresponds to the rest of the computation. This technique is frequently used in specifying the semantics of concurrency (Mosses, 1990; de Bakker and de Vink, 1996), however the approach that we use here makes no assumption of what the atomic steps are.

```
data R m a = Computed a | Resume (m (R m a))

instance Monad m => Monad (R m) where
   return           = Computed
   Computed v >>= f  = f v
   Resume m >>= f    = Resume (m >>= \r -> return (r >>= f))
```

Parameter `m` is a monad representing the resumptionless computations, i.e. the atomic steps. A computation of type `R m a` is either a computed value of type `a` or a computation of type `m (R m a)`, which produces a resumption.[12]

Before we can proceed with the definition of `M`, a few operations and properties of `R` must be defined. Functions `runR` and `stepR` convert between computations of type `R m a` and `m a` in both directions. The first fully evaluates a resumption by performing all atomic steps. The second produces a computation with just one atomic step.

```
runR :: Monad m => R m a -> m a
runR (Computed v) = return v
runR (Resume m)   = m >>= runR

stepR :: Monad m => m a -> R m a
stepR m = Resume (m >>= (return . Computed))
```

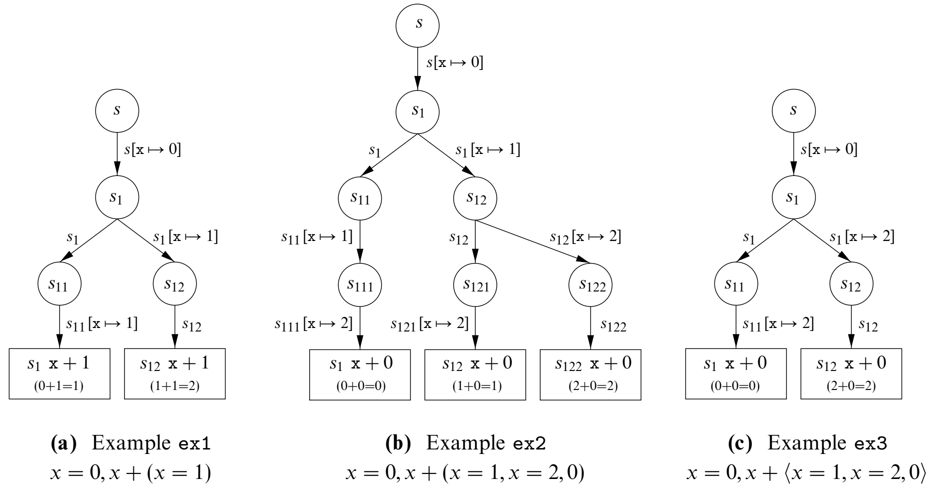
Using `stepR` it is possible to introduce state operations in resumption monads, provided that they are supported by the corresponding resumptionless monads. Each state operation corresponds to one atomic step in the interleaved computation.

```
instance StateMonad s m => StateMonad s (R m) where
   setState = stepR . setState
```

A new version of monad `M` which allows interleaved computations can be defined by applying the monad transformer `R` to the direct semantics monad that has been used in the previous section. We retain the list monad for the implementation of

[12] The resumption monad transformer is usually defined in literature as $R\,m\,a = m(a + R\,m\,a)$, e.g. in the work of Espinosa (1995). The approach taken here corresponds to $R\,m\,a = a + m(R\,m\,a)$ and allows constant computations requiring no atomic steps.

https://doi.org/10.1017/S095679680000366X Published online by Cambridge University Press



(a) Example `ex1`
$x = 0, x + (x = 1)$

(b) Example `ex2`
$x = 0, x + (x = 1, x = 2, 0)$

(c) Example `ex3`
$x = 0, x + \langle x = 1, x = 2, 0 \rangle$

Fig. 1. Three examples of interleaving semantics.

multiple results.

```
type M a = R (D S []) a
```

Again, it is necessary to redefine how monad `M` is an instance of class `StrongMonad`, and one more property of `R` is specified for this purpose. On condition that the resumptionless monad `m` supports multiple results, the resumption monad `R m` allows interleaving, as defined below. If any of the operands of `+:+` requires no atomic steps, we only need to evaluate the other operand. If both computations require atomic steps, operator `+:+` must choose between performing an atomic step from the first operand or from the second.

```
instance MultiMonad m => StrongMonad (R m) where
   Computed v1 +:+ r2 =  r2 >>= \v2 -> return (v1, v2)
   r1 +:+ Computed v2 =  r1 >>= \v1 -> return (v1, v2)
   r1@(Resume m1) +:+ r2@(Resume m2) = Resume(
      (m1 >>= \r1' -> return (r1' +:+ r2)) +|+
      (m2 >>= \r2' -> return (r1 +:+ r2')))
```

Returning to our examples, the resumption semantics of $x = 0, x + (x = 1)$ produces 1 or 2, exactly as in the 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, if the (implicit) dereferencing of x is interleaved between atomic steps $x = 1$ and $x = 2$. All possible evaluation paths for both examples are shown in schematic form in figures 1(a) and (b), with emphasis on state transitions rather than on evaluated results.

```
? sem ex1
[(1,state),(2,state)]
? sem ex2
[(0,state),(1,state),(2,state)]
```

https://doi.org/10.1017/S095679680000366X Published online by Cambridge University Press

4.4 Evaluation in One Atomic Step

It is often useful to disable interleaving and evaluate an arbitrary expression in a single atomic step. To this end, we introduce a new construct in ELSE. The revised abstract syntax is shown below.

$$E \,:\, \mathbf{Expr} \;\;::=\;\; n \;|\; I \;|\; I{=}E \;|\; E_1{+}E_2 \;|\; -E \;|\; E_1{,}E_2 \;|\; \langle E \rangle$$

Expression $\langle E \rangle$ 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 required changes in the syntax and semantics of ELSE are shown in the following Haskell code. The semantics of the newly introduced construct may be expressed using `stepR` and `runR`. A computation consisting of several steps is converted to an equivalent one consisting of a single step.

```
data Expr = E_int N | E_ide Ide | E_assign (Ide, Expr) | E_plus (Expr, Expr)
          | E_neg Expr | E_comma (Expr, Expr) | E_unit Expr

sem (E_unit e) = stepR (runR (sem e))
```

The following example, corresponding to expression $x = 0, x + \langle x = 1, x = 2, 0 \rangle$, illustrates the use of our new construct. It is similar to $x = 0, x + (x = 1, x = 2, 0)$ with the only difference that the right operand of $+$ is evaluated in a single atomic step.

```
ex3 = E_comma (E_assign ("x", E_int 0),
               E_plus (E_ide "x",
                       E_unit (E_comma (E_assign ("x", E_int 1),
                               E_comma (E_assign ("x", E_int 2),
                               E_int 0)))))
```

As shown below, evaluation of this expression cannot produce the result 1, since the (implicit) dereferencing of x cannot take place in between assignments $x = 1$ and $x = 2$. The schematic evaluation of this expression is shown in figure 1 (c).

```
? sem ex3
[(0,state),(2,state)]
```

4.5 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.[13] 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. 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.

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

https://doi.org/10.1017/S095679680000366X Published online by Cambridge University Press

The C standard is very careful in its attempt not to overspecify the semantics of the language, to allow reasonable optimizations in implementations. 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 no side effects of subsequent evaluations shall have taken place. (ANSI, 1990, §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. According to the ANSI C standard, sequence points are located:

- after the evaluation of a full expression,
- before a function call but after the evaluation of actual parameters, and
- after the evaluation of the left operand of operators `&&`, `||`, `,` (comma) and `?:` (conditional).

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. (ANSI, 1990, §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 $x = x+1$. However, we should notice 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 involving the same global variables, since the order in which they will be called is not specified by the standard.

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 $\langle \# E \rangle$ is reasonably equivalent to a call to a C function with no arguments. The revised abstract syntax is given below.

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

The required changes in the syntax and semantics of ELSE are given in the following Haskell code. Function `seqpt :: M ()` represents a computation that enforces a sequence point and returns no interesting result. The semantics degenerates into that of the previous section if we take `seqpt = return ()`.

```
data Expr = E_int N | E_ide Ide | E_assign (Ide, Expr) | E_plus (Expr, Expr)
          | E_neg Expr | E_comma (Expr, Expr) | E_unit Expr | E_call Expr
```

https://doi.org/10.1017/S095679680000366X Published online by Cambridge University Press

```
sem (E_comma (e1, e2)) =
   sem e1 >>= \n ->
   seqpt >>= \u ->
   sem e2
sem (E_call e) =
   seqpt >>= \u1 ->
   sem e >>= \n ->
   seqpt >>= \u2 ->
   return n
```

To distinguish between a side effect that has taken place and one that has not, it is necessary to change the definition of state. It is possible to define `S` as a pair of two functions, the first representing the memory and the second representing the pending side effects. Based on the restrictions that at most one writing to a memory location takes place between two successive sequence points and that no reading takes place after the writing, we may assume in our semantics that all side effects take place at the following sequence point by a call to `commit :: S -> S`. The new definition of `S` is given below.

```
newtype S = S (Ide -> N, Ide -> Maybe N)

instance State S where
   initial = S (\i -> error ("Variable " ++ i ++ " has not been initialized"),
                \i -> Nothing)
   load i (S (sn, sx)) =
      case sx i of
         Nothing -> sn i
         _       -> error ("Store-load conflict for variable " ++ i)
   store i n (S (sn, sx)) =
      case sx i of
         Nothing -> S (sn, update i (Just n) sx)
         _       -> error ("Store-store conflict for variable " ++ i)
```

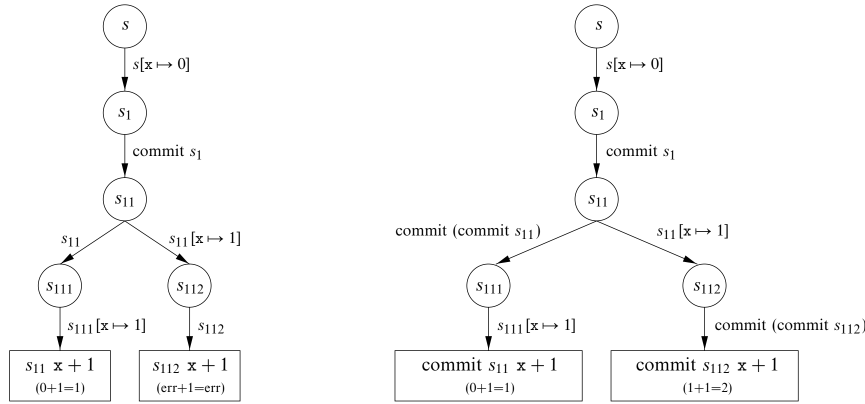
The `Maybe` monad, defined in Haskell's Standard Prelude and repeated in the appendix, is used in the second element of the pair and the element `Nothing` indicates that there are no side effects pending for a given variable. The implementations of `load` and `store` guarantee that no reading or writing is permitted if a side effect for the same variable is pending. Subsequently, we can define functions `commit` and `seqpt` as follows:

```
commit :: S -> S
commit (S (sn, sx)) = S (\i -> case sx i of
                                  Nothing -> sn i
                                  Just v  -> v,
                         \i -> Nothing)

seqpt :: M ()
seqpt = setState commit >>= \s -> return ()
```

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 `store` 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:

https://doi.org/10.1017/S095679680000366X Published online by Cambridge University Press



(a) Example `ex1`
$x = 0, x + (x = 1)$

(b) Example `ex4`
$x = 0, \langle \# x \rangle + (x = 1)$

Fig. 2. Two examples of interleaving semantics with sequence points.

- 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 `load`.
- A read access that does not determine the new value is only allowed by the definition of `load` if it precedes the write access in the evaluation order. However, 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 `load` produces an error. This error will propagate in the semantics of expressions. 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 a first example, we consider again the expression $x = 0, x + (x = 1)$. The semantics shown in figure 2(a) is similar to that in figure 1(a), with the exception of the extra `commit` step. We notice, however, that the result 2 has been replaced by an error in the evaluation of this expression, since read access $s_{112}\ x$ is performed on state $s_{11}[x \mapsto 1]$ without an intermediate sequence point.

```
? sem ex1
[(1,state),(
Program error: Store-load conflict for variable x
```

A similar error results from the evaluation of $x = 0, x + (x = 1, x = 2, 0)$ as well. However, expression $x = 0, x + \langle x = 1, x = 2, 0 \rangle$ still produces the results 0 and 2, since the evaluation of the right operand of $+$ does not leave pending side-effects, because of the sequence points enforced by the two comma operators.

Two more examples are given below. The first is illustrated in figure 2(b) and corresponds to the expression $x = 0, \langle \# x \rangle + (x = 1)$. The second corresponds to the expression $x = 0, \langle \# x \rangle + (x = 1, x = 2, 0)$.

https://doi.org/10.1017/S095679680000366X Published online by Cambridge University Press

```
ex4 = E_comma (E_assign ("x", E_int 0),
               E_plus (E_unit (E_call (E_ide "x")), E_assign ("x", E_int 1)))

ex5 = E_comma (E_assign ("x", E_int 0),
               E_plus (E_unit (E_call (E_ide "x")),
                       E_comma (E_assign ("x", E_int 1),
                       E_comma (E_assign ("x", E_int 2),
                                E_int 0))))
```

They are variations of the first two examples `ex1` and `ex2`, only in both cases the left operand of + is protected as if it was wrapped by a function call. No errors result from their evaluation. Notice, however, that in the second case the results 1 and 2 appear twice, because of the presence of additional `commit` steps.

```
? sem ex4
[(1,state),(2,state)]
? sem ex5
[(0,state),(1,state),(1,state),(2,state),(2,state)]
```

5 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 (1982) and Kowaltowski (1977) 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 (1996) 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 (1990) and Bloss (1994). 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 have been formally specified in literature, at least partially. Unspecified evaluation order is modelled 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 (1980) addresses the semantics of pre-ANSI C, using the denotational approach and assuming left-to-right evaluation of expressions. In the work of Gurevich and Huggins (1993) 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 4.2 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 (1996), which removes side effects

https://doi.org/10.1017/S095679680000366X Published online by Cambridge University Press

from expressions and treats them as separate statements. In the work of Cook and Subramanian (1994), 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.* (1994) have also developed a denotational semantics for C based on temporal logic. 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 (1997; 1998) in the form of operational semantics with small-step reductions. No similar denotational approach is known to us.

6 Conclusion

The main contribution of this paper is an exploration of the denotational semantics of evaluation order in eager expression languages with side effects. The semantics of a simple example language is developed under various different evaluation strategies in a unified way, based on monads and monad transformers which improve the modularity and elegance of the result. By using a dialect of the functional programming language Haskell as a metalanguage, instead of mathematical notation, the developed semantics can be easily tested and evaluated. Taking into consideration that the intended semantics of some of the studied evaluation strategies is quite messy, it is also argued that the simplicity and conciseness of the result helps our understanding of evaluation order in general.

Although the example language ELSE has been kept as primitive as possible, the semantics of additional programming constructs (e.g. iteration, functions, variable aliasing, exceptions) can be modelled easily, using the same modular framework. The direct semantics monad transformer in the definition of `M` can be easily replaced by a continuation monad transformer, which is needed in a complete formal treatment of C. It is also easy to extend operator `+:+` for any number of operands. 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 and a significant step towards a complete and accurate denotational description of ANSI C.

A Additional code

Additional code that is necessary for executing the developed semantics in the Haskell environment is given in this appendix. Also, for the sake of completeness, some definitions from Haskell's standard prelude are repeated here.

Identity function (repeated)

```
id :: a -> a
id x = x
```

https://doi.org/10.1017/S095679680000366X Published online by Cambridge University Press

Update function

```
update :: Eq a => a -> b -> (a -> b) -> a -> b
update i n f j = if i == j then n else f i
```

List monad (repeated)

```
instance Monad [] where
   return v      = [v]
   []     >>= f  = []
   (v:vs) >>= f  = f v ++ (vs >>= f)
```

Maybe monad (repeated)

```
data Maybe a = Just a | Nothing

instance Monad Maybe where
   return         =  Just
   Just x  >>= k  =  k x
   Nothing >>= k  =  Nothing
```

Printing states and computations

```
instance Show S where
   showsPrec d s = ("state" ++)

instance Show a => Show (Id a) where
   showsPrec d (Id x) = showsPrec d x

instance (State s, Show (m (a, s))) => Show (D s m a) where
   showsPrec d (D r) = showsPrec d (r initial)

instance (Monad m, Show a, Show (m a)) => Show (R m a) where
   showsPrec d (Computed v) = showsPrec d v
   showsPrec d (Resume m)   = showsPrec d (m >>= runR)
```

References

ANSI (1990) *ANSI/ISO 9899-1990, American national standard for programming languages: C*. American National Standards Institute, New York, NY. Revision and redesignation of ANSI X3.159-1989.

Black, P. E. and Windley, P. J. (1996) Inference rules for programming languages with side effects in expressions. *Proc. 9th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'96)*, pp. 51–60. Turku, Finland. Springer-Verlag.

Bloss, A. (1994) Path analysis and the optimization of nonstrict functional languages. *ACM Trans. Programming Lang. and Syst.*, **16**(3), 328–369.

Boehm, H. J. (1982) A logic for expressions with side effects. *Proc. ACM Symposium on Principles of Programming Languages*, pp. 268–280.

Cook, J. and Subramanian, S. (1994) A formal semantics for C in Nqthm. *Technical Report 517D*, Trusted Information Systems.

Cook, J., Cohen, E. and Redmond, T. (1994) A formal denotational semantics for C. *Technical Report 409D*, Trusted Information Systems.

de Bakker, J. and de Vink, E. (1996) *Control Flow Semantics.* Foundations of Computing Series. MIT Press.

https://doi.org/10.1017/S095679680000366X Published online by Cambridge University Press

Draghicescu, M. and Purushothaman, S. (1990) A compositional analysis of evaluation-order and its application. *Conference Record of the ACM Symposium on Lisp and Functional Programming*, pp. 242–250.

Espinosa, D. A. (1995) Semantic Lego. *PhD thesis*, Columbia University, Department of Computer Science.

Filinski, A. (1996) Controlling effects. *PhD thesis*, Carnegie Mellon University, School of Computer Science. (Also as Technical Report CMU-CS-96-119.)

Gurevich, Y. and Huggins, J. K. (1993) The semantics of the C programming language. In: Börger, E. *et al.* (eds.), *Selected papers from CSL'92: Lecture Notes in Computer Science*, **702**, pp. 274–308. Springer-Verlag.

Jones, M. P. (1994) The implementation of the Gofer functional programming system. *Research Report YALEU/DCS/RR-1030*, Yale University, Department of Computer Science.

Jones, M. P. (1995) A system of constructor classes: Overloading and implicit higher-order polymorphism. *J. Functional Programming*, **5**(1), 1–37.

Kowaltowski, T. (1977) Axiomatic approach to side effects and general jumps. *Acta informatica*, **7**, 357–360.

Liang, S. (1998) Modular monadic semantics and compilation. *PhD thesis*, Yale University, Department of Computer Science.

Liang, S., Hudak, P. and Jones, M. (1995) Monad transformers and modular interpreters. *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95).*

Moggi, E. (1990) An abstract view of programming languages. *Technical Report ECS-LFCS-90-113*, University of Edinburgh, Laboratory for Foundations of Computer Science.

Mosses, P. D. (1990) Denotational semantics. In: van Leeuwen, J. (ed.), *Handbook of Theoretical Computer Science*, vol. B, pp. 577–631. Elsevier.

Norrish, M. (1997) An abstract dynamic semantics for C. *Technical Report TR-421*, University of Cambridge, Computer Laboratory.

Norrish, M. (1998)) C formalized in HOL. *PhD thesis*, University of Cambridge, Computer Laboratory.

Papaspyrou, N. S. (1998) A formal semantics for the C programming language. *PhD thesis*, National Technical University of Athens, Software Engineering Laboratory.

Peyton Jones, S. and Hughes, J. (eds.) (1999) *Report on the programming language Haskell 98: A non-strict purely functional language.* Available from `http://haskell.org/`.

Peyton Jones, S., Peyton Jones, M. and Meijer, E. (1997) *Type classes: An exploration of the design space*. Haskell Workshop. Available from Haskell Workshop's home page at `http://www.cse.ogi.edu/~jl/ACM/Haskell.html`.

Sethi, R. (1980) A case study in specifying the semantics of a programming language. *Proc. 7th Annual ACM Symposium on Principles of Programming Languages*, pp. 117–130.

Wadler, P. (1992) The essence of functional programming. *Proc. 19th Annual Symposium on Principles of Programming Languages (POPL'92).*

https://doi.org/10.1017/S095679680000366X Published online by Cambridge University Press