Under consideration for publication in the Journal of Functional Programming

Programming Language Semantics

It's Easy As 1,2,3

GRAHAM HUTTON
School of Computer Science, University of Nottingham, UK
(email: `graham.hutton@nottingham.ac.uk`*)*

Abstract

Programming language semantics is an important topic of theoretical computer science, but one that beginners often find challenging. This article provides a tutorial introduction to the subject, in which the language of integers and addition is used as a minimal setting in which to present a range of semantic concepts in simple manner. In this setting, it's easy as 1,2,3.

1 Introduction

When studying a new concept, it is often beneficial to begin with a simple example to understand the basic ideas. This article is about such an example that can be used to present a range of topics in programming language semantics: the language of simple arithmetic expressions built up from integers values using an addition operator.

This language has played a key role in my own work for many years. In the beginning, it was used to help *explain* semantic ideas, but over time it also became a mechanism to help *discover* new ideas, and has featured in many of my publications. The purpose of this article is to consolidate this experience, and show how the language of integers and addition can be used to present a range of semantic concepts in a simple manner.

Using a minimal language to explore semantic ideas is an example of Occam's Razor (Duignan, 2018), a philosophical principle that favours the simplest explanation for a phenomenon. While the language of integers and addition does not provide features that are necessary for actual programming, it *does* provide just enough structure to explain many concepts from semantics. In particular, the integers provide a simple notion of 'value', and the addition operator provides a simple notion of 'computation'. This language has been used by many authors in the past, such as McCarthy and Painter (1967), Wand (1982) and Wadler (1998), to name but a few. However, this article is the first time the language has been used as a general tool for exploring different semantic topics.

Of course, one could consider a more sophisticated minimal language, such as a simple imperative language with mutable variables, or a simple functional language based on the lambda calculus. However, doing so then brings in other concepts such as stores, environments, substitutions, variable capture, and so on. Learning about these is important, but my experience time and time again is that there is much to be gained by *first* focusing on the simple language of integers and addition. Once the basic ideas are developed and

ZU064-05-FPR paper 20 December 2022 11:2

understood in this setting, one can then extend the language with other features of interest, an approach that has proved useful in many aspects of my own work.

The article is written in a tutorial style that does not assume prior knowledge of semantics, and is aimed at the level of advanced undergraduates and beginning PhD students. Nonetheless, I hope that experienced readers will also find useful ideas and inspiration for their own work. Beginners may wish to initially focus on sections 2–7, which introduce and compare a number of widely-used approaches to semantics (denotational, small-step, contextual and big-step), together with the proof technique of rule induction. In turn, those with more experience may wish to proceed quickly through to section 8, which presents an extended example of how the simple expression language can be used to help discover new semantic ideas (transforming semantics into implementations.)

Note that the article does not aim to provide a comprehensive account of semantics in either breadth or depth, but rather to summarise the basic ideas and benefits of our minimal approach, and provide pointers to further reading. Haskell is used throughout as a metalanguage to implement semantic ideas, which helps to make the ideas more concrete and allows them to be executed. All the code is available online as supplementary material.

2 Arithmetic Expressions

We begin by defining our language of interest, namely simple arithmetic expressions built up from the set $\mathbb{Z}$ of integer values using the addition operator $+$. Formally, the language E of such expressions is defined by the following context-free grammar:

$$E \ ::= \ \mathbb{Z} \mid E + E$$

That is, an expression is either an integer value or the addition of two sub-expressions. We assume that parentheses can be freely used as required to disambiguate expressions written in normal textual form, such as $1+(2+3)$. The grammar for expressions can also be translated directly into a Haskell datatype declaration, for which purpose we use the built-in type *Integer* of arbitrary precision integers:

```
data Expr = Val Integer | Add Expr Expr
```

For example, the expression $1+2$ is represented by the term *Add* (*Val* 1) (*Val* 2). From now on, we mainly consider expressions represented in Haskell.

3 Denotational Semantics

In the first part of the article we show how our simple expression language can be used to explain and compare a number of different approaches to specifying the semantics of languages. In this section we consider the *denotational* approach to semantics (Scott & Strachey, 1971), in which the meaning of terms in a language is defined using a valuation function that maps terms into values in an appropriate semantic domain.

Formally, a denotational semantics for a language T of syntactic terms comprises two components: a set V of *semantic values*, and a *valuation function* of type $T \to V$ that maps terms to their meaning as values. The valuation function is typically written by enclosing a term in *semantic brackets*, writing $[\![t]\!]$ for result of applying the valuation function to the term t. The semantic brackets are also known as Oxford or Strachey brackets, after the pioneering work of Christopher Strachey on the denotational approach.

In addition to the above, the valuation function is required to be *compositional*, in the sense that the meaning of a compound term is defined purely in terms of the meaning of its subterms. Compositionality aids understanding by ensuring that the semantics is modular, and also has the important technical benefit that it supports the use of structural induction to reason about the semantics. When the set of semantic values is clear, a denotational semantics is often identified with the underlying valuation function.

Arithmetic expressions of type *Expr* have a particularly simple denotational semantics, given by taking V as the Haskell type *Integer* of integers, and defining an evaluation function of type $Expr \to Integer$ by the following two equations:

$$\begin{aligned} [\![Val\, n]\!] &= n \\ [\![Add\, x\, y]\!] &= [\![x]\!] + [\![y]\!] \end{aligned}$$

The first equation states that the value of an integer is simply the integer itself, while the second states that the value of an addition is given by adding together the values of its two sub-expressions. This definition manifestly satisfies the compositionality requirement, because the meaning of a compound expression $Add\, x\, y$ is defined purely by applying the $+$ operator to the meanings of the two sub-expressions x and y.

The evaluation function can also be translated directly into a Haskell function definition, by simply rewriting the mathematical definition in Haskell notation:

$$\begin{aligned} &eval :: Expr \to Integer \\ &eval\ (Val\, n) \quad\ = n \\ &eval\ (Add\, x\, y) = eval\, x + eval\, y \end{aligned}$$

More generally, a denotational semantics can be viewed as an evaluator (or interpreter) that is written in a functional language. For example, using the above definition we have $eval\ (Add\ (Val\, 1)\ (Add\ (Val\, 2)\ (Val\, 3))) = 1 + (2 + 3) = 6$, or pictorially:

$$eval \left(\begin{tikzcd} & Add \arrow[dl] \arrow[dr] & & \\ Val\, 1 & & Add \arrow[dl] \arrow[dr] & \\ & Val\, 2 & & Val\, 3 \end{tikzcd} \right) = \begin{tikzcd} & + \arrow[dl] \arrow[dr] & & \\ 1 & & + \arrow[dl] \arrow[dr] & \\ & 2 & & 3 \end{tikzcd} = 6$$

From this example, we see that an expression is evaluated by replacing each *Add* constructor by the addition function $+$ on integers, and by removing each *Val* constructor, or equivalently, by replacing each *Val* by the identity function *id* on integers. That is, even though *eval* is defined recursively, because the semantics is compositional its behaviour can be understood as simply replacing the constructors for expressions by other functions. In this manner, a denotational semantics can also be viewed as an evaluation function that is defined by 'folding' over the syntax of the source language:

$$\begin{aligned} &eval :: Expr \to Integer \\ &eval = fold\ id\ (+) \end{aligned}$$

The fold operator (Meijer *et al.*, 1991) captures the idea of replacing the constructors of the language by other functions, here replacing *Val* and *Add* by functions f and g:

$$\begin{aligned}
&\mathit{fold} :: (\mathit{Integer} \to a) \to (a \to a \to a) \to \mathit{Expr} \to a\\
&\mathit{fold}\, f\; g\; (\mathit{Val}\, n) \quad\;\; = f\; n\\
&\mathit{fold}\, f\; g\; (\mathit{Add}\, x\, y) = g\; (\mathit{fold}\, f\; g\; x)\; (\mathit{fold}\, f\; g\; y)
\end{aligned}$$

Note that a semantics defined using *fold* is compositional by definition, because the result of folding over an expression *Add x y* is defined purely by applying the given function g to the result of folding over the two argument expressions x and y.

We conclude this section with two further remarks. First of all, if we had chosen the grammar $E ::= \mathbb{Z} \mid E + E$ as our source language, rather than the type *Expr*, then the denotational semantics would have the following form:

$$\begin{aligned}
[\![n]\!] \quad\;\; &= n\\
[\![x + y]\!] &= [\![x]\!] + [\![y]\!]
\end{aligned}$$

However, in this version the same symbol $+$ is now used for two different purposes: on the left side it is a *syntactic* constructor for building terms, while on the right side it is a *semantic* operator for adding integers. We avoid such issues and keep a clear distinction between syntax and semantics by using the type *Expr* as our source language, which provides special-purpose constructors *Val* and *Add* for building expressions.

And secondly, note that the above semantics for arithmetic expressions does not specify the order of evaluation, that is, the order in which the two arguments of addition should be evaluated. In this case the order has no effect on the final value, but if we did wish to to make evaluation order explicit this requires the introduction of additional structure into the semantics, which we will discuss later on when we consider abstract machines.

Further reading The standard reference on denotational semantics is Schmidt (1986), while Winskel's (1993) textbook on formal semantics provides a concise introduction to the approach. The problem of giving a denotational semantics for the lambda calculus, in particular the technical issues that arise with recursively defined functions and types, led to the development of domain theory (Abramsky & Jung, 1994).

The idea of defining denotational semantics using fold operators is explored further in (Hutton, 1998). The simple integers and addition language has also been used as a basis for studying a range of other language features, including exceptions (Hutton & Wright, 2004), interrupts (Hutton & Wright, 2007), transactions (Hu & Hutton, 2009), non-determinism (Hu & Hutton, 2010) and state (Bahr & Hutton, 2015).

4 Small-Step Semantics

Another popular approach to semantics is the *operational* approach (Plotkin, 1981), in which the meaning of terms is defined using an execution relation that specifies how terms can be executed in an appropriate machine model. There are two basic forms of operational semantics: *small-step*, which describes the individual steps of execution, and *big-step*, which describes the overall results of execution. In this section we consider the small-step approach, which is also known as 'structural operational semantics' and will return to the big-step approach later on in Section 6.

Formally, a small-step operational semantics for a language T of syntactic terms comprises two components: a set S of *execution states*, and a *transition relation* on S that relates each state to all states that can be reached by performing a single execution step. If two states s and s' are related, we say that there is a transition from s to s', and write this as $s \longrightarrow s'$. More general notions of transition relation are sometimes used, but this simple notion suffices for our purposes here. When the set of states is clear, an operational semantics is often identified with the underlying transition relation.

Arithmetic expressions have a simple small-step operational semantics, given by taking S as the Haskell type *Expr* of expressions, and defining the transition relation on *Expr* by the following three inference rules:

$$\frac{}{\mathit{Add}\ (\mathit{Val}\ n)\ (\mathit{Val}\ m)\ \longrightarrow\ \mathit{Val}\ (n+m)}$$

$$\frac{x\ \longrightarrow\ x'}{\mathit{Add}\ x\ y\ \longrightarrow\ \mathit{Add}\ x'\ y} \qquad\qquad \frac{y\ \longrightarrow\ y'}{\mathit{Add}\ x\ y\ \longrightarrow\ \mathit{Add}\ x\ y'}$$

The first rule states that two values can be added together to give a single value, while the last two rules permit transitions to be made on the left and right sides of an addition. Note that the semantics is non-deterministic, because an expression could have more than one possible transition. For example, the expression $(1+2)+(3+4)$, written here in normal syntax for brevity, has two possible transitions, because the first rule can be applied on either side of the top-level addition using the second and third rules:

$$\begin{aligned}(1+2)+(3+4) &\ \longrightarrow\ 3+(3+4)\\ (1+2)+(3+4) &\ \longrightarrow\ (1+2)+7\end{aligned}$$

By repeated application of the transition relation, we can generate a transition tree that captures all possible execution paths for an expression. For example, the expression above gives rise to the following tree, which captures the two possible execution paths:

$$\begin{tikzcd}
& (1+2)+(3+4) \arrow[dl] \arrow[dr] & \\
3+(3+4) \arrow[d] & & (1+2)+7 \arrow[d] \\
3+7 \arrow[d] & & 3+7 \arrow[d] \\
10 & & 10
\end{tikzcd}$$

The transition relation can also be translated into a Haskell function definition, by exploiting the fact that a relation can be represented as a non-deterministic function that returns all possible values that are related to a given value. Using the list comprehension notation, it is straightforward to define a function that returns the list of all expressions that can be reached from a given expression by performing a single transition:

```
trans :: Expr → [Expr]
trans (Val n)            = []
```

$$\begin{aligned}
&\mathit{trans}\ (\mathit{Add}\ (\mathit{Val}\ n)\ (\mathit{Val}\ m)) = [\mathit{Val}\ (n+m)] \\
&\mathit{trans}\ (\mathit{Add}\ x\ y) \qquad\qquad\quad = [\mathit{Add}\ x'\ y \mid x' \leftarrow \mathit{trans}\ x] +\!\!+ [\mathit{Add}\ x\ y' \mid y' \leftarrow \mathit{trans}\ y]
\end{aligned}$$

In turn, we can define a Haskell datatype for transition trees, and an execution function that converts expressions into trees by repeated application of the transition function:

$$\begin{aligned}
&\mathbf{data}\ \mathit{Tree}\ a = \mathit{Node}\ a\ [\mathit{Tree}\ a] \\
&\mathit{exec} :: \mathit{Expr} \to \mathit{Tree}\ \mathit{Expr} \\
&\mathit{exec}\ e = \mathit{Node}\ e\ [\mathit{exec}\ e' \mid e' \leftarrow \mathit{trans}\ e]
\end{aligned}$$

From this definition, we see that an expression is executed by taking the expression itself as the root of the tree, and generating a list of residual expressions to be processed to give the subtrees by applying the *trans* function. That is, even though *exec* is defined recursively, its behaviour can be understood as simply applying the identity function to give the root of the tree, and the transition function to generate a list of residual expressions to be processed to give the subtrees. In this manner, a small-step operational semantics can be viewed as giving rise to an execution function that is defined by 'unfolding' to transition trees:

$$\begin{aligned}
&\mathit{exec} :: \mathit{Expr} \to \mathit{Tree}\ \mathit{Expr} \\
&\mathit{exec} = \mathit{unfold}\ \mathit{id}\ \mathit{trans}
\end{aligned}$$

The unfold operator (Gibbons & Jones, 1998) captures the idea of generating a tree from a seed value x by applying a function f to give the root, and a function g to give a list of residual values that are then processed in the same way to produce the subtrees:

$$\begin{aligned}
&\mathit{unfold} :: (a \to b) \to (a \to [a]) \to a \to \mathit{Tree}\ b \\
&\mathit{unfold}\ f\ g\ x = \mathit{Node}\ (f\ x)\ [\mathit{unfold}\ f\ g\ x' \mid x' \leftarrow g\ x]
\end{aligned}$$

In summary, whereas denotational semantics corresponds to 'folding over syntax trees', operational semantics corresponds to 'unfolding to transition trees'. Thinking about semantics in terms of recursion operators reveals a duality that might otherwise have been missed, and still isn't as widely known as it should be.

We conclude with three further remarks. First of all, note that if the original grammar for expressions was used as our source language rather than the type *Expr*, then the first inference rule for the semantics would have the following form:

$$\frac{}{n+m \ \longrightarrow\ n+m}$$

However, this rule would be rather confusing unless we introduced some additional notation to distinguish the syntactic $+$ on the left side from the semantic $+$ on the right side, which is precisely what is achieved by the use of the *Expr* type.

Secondly, the above semantics for expressions does not specify the order of evaluation, or more precisely, it captures *all* possible evaluation orders. However, if we do wish to specify a particular evaluation order, it is straightforward to modify the inference rules to achieve this. For example, replacing the second *Add* rule by the following would ensure the first argument to addition is always evaluated before the second:

$$\frac{y \ \longrightarrow\ y'}{\mathit{Add}\ (\mathit{Val}\ n)\ y \ \longrightarrow\ \mathit{Add}\ (\mathit{Val}\ n)\ y'}$$

In contrast, as noted in the previous section, making evaluation order explicit in a denotational semantics is more challenging. Being able to specify evaluation order in a straightforward manner in an important benefit of the small-step approach.

And finally, using Haskell as our meta-language the transition relation was implemented in an indirect manner as a non-deterministic function, in which the ordering of the equations is important because the patterns used are not disjoint. In contrast, if we used a meta-language with dependent types, such as Agda (Norell, 2007), the transition relation could be implemented directly as an *inductive family* (Dybjer, 1994), with no concerns about ordering in the definition. However, we chose to use Haskell rather than a more sophisticated language in order to make the ideas as widely accessible as possible. Nonetheless, it is important to acknowledge the limitations of this choice.

Further reading The origins of the operational approach to semantics are surveyed in (Plotkin, 2004). In this article we primarily focus our attention on denotational and operational approaches to semantics, but there are a variety of other approaches too, including axiomatic (Hoare, 1969), algebraic (Goguen & Malcolm, 1996), action (Mosses, 2005) and game (Abramsky & McCusker, 1999) semantics.

The idea of defining operational semantics using unfold operators, and the duality with denotational semantics defined using fold operators, is explored in (Hutton, 1998).

5 Contextual Semantics

When defining a small-step semantics there are usually a number of basic rules that capture the core behaviour of the language features, with the remainder being 'structural' rules that express how the basic rules can be applied in larger terms. For example, the semantics in the previous section has one basic rule for adding values, and two structural rules that allow addition to be performed in larger expressions. Separating these two forms of rules gives rise to the notion of *contextual* semantics (Felleisen & Hieb, 1992).

Informally, a context in this setting is a term with a 'hole', usually written as [–], which can be 'filled' with another term later on. In a contextual semantics, the hole represents the location where a single basic step of execution may take place within a term. For example, consider the following transition from the previous section:

$$(1+2)+(3+4) \quad \longrightarrow \quad 3+(3+4)$$

In this case, an addition is performed on the left side of the term. This idea can be made precise by saying that we can perform the basic step $1+2 \longrightarrow 3$ in the context $[–]+(3+4)$, in which the hole indicates where the addition takes place. For arithmetic expressions, the language C of contexts can formally be defined by the following grammar:

$$C \ ::= \ [–] \mid C+E \mid E+C$$

That is, a context is either a hole, or a context on either side of the addition of an expression. As previously, however, to keep a clear distinction between syntax and semantics we translate the grammar into a Haskell datatype declaration:

```
data Con = Hole | AddL Con Expr | AddR Expr Con
```

Using this type, it is then straightforward to define what it means to fill the hole in a context c with a given expression x, which we write as $c\,[x]$:

$$\begin{array}{lcl}
\mathit{Hole}\ \ [x] & = & x \\
(\mathit{AddL}\ c\ e)\ [x] & = & \mathit{Add}\ (c\,[x])\ e \\
(\mathit{AddR}\ e\ c)\ [x] & = & \mathit{Add}\ e\ (c\,[x])
\end{array}$$

That is, if the context is a hole we simply return the given expression, otherwise we recurse on the left or right side of an addition as appropriate. Note that the above is a mathematical definition for hole filling, which uses Haskell syntax for contexts and expressions. As usual, we'll see shortly how it can be implemented in Haskell itself.

Using the idea of hole filling, we can now redefine the small-step semantics for expressions in contextual style, by means of the following two inference rules:

$$\frac{}{\mathit{Add}\ (\mathit{Val}\ n)\ (\mathit{Val}\ m)\ \rightarrowtail\ \mathit{Val}\ (n+m)} \qquad \frac{x\ \rightarrowtail\ x'}{c\,[x]\ \longrightarrow\ c\,[x']}$$

The first rule defines a reduction relation $\rightarrowtail$ that captures the basic behaviour of addition, while the second defines a transition relation $\longrightarrow$ that allows the first rule to be applied in any context, that is, to either argument of an addition. In this manner, we have now refactored the small-step semantics into a single basic rule and a single structural rule. Moreover, if we subsequently wished to extend the language with other features, this usually only requires adding new basic rules and extending the notion of contexts, but typically does not require adding new structural rules.

The contextual semantics can readily be translated into Haskell. Defining hole filling is just a matter of rewriting the mathematical definition in Haskell syntax:

```
fill :: Con → Expr → Expr
fill Hole         x = x
fill (AddL c e) x = Add (fill c x) e
fill (AddR e c) x = Add e (fill c x)
```

In turn, the dual operation, which splits an expression into all possible pairs of contexts and expressions, can be defined using the list comprehension notation:

```
split :: Expr → [(Con, Expr)]
split e = (Hole, e) : case e of
                Val n     → []
                Add x y → [(AddL c y, e) | (c, e) ← split x] ++
                              [(AddR x c, e) | (c, e) ← split y]
```

The behaviour of this definition can be formally characterised as follows: a pair (c, x) comprising a context c and an expression x is an element of the list returned by *split e* precisely when $\mathit{fill}\ c\ x = e$. Using these two functions, the contextual semantics can then be translated into Haskell function definitions that return the lists of all expressions that can be reached by performing a single reduction step,

```
reduce :: Expr → [Expr]
reduce (Add (Val n) (Val m)) = [Val (n + m)]
reduce _ = []
```

or a single transition step:

$trans :: Expr \to [Expr]$
$trans\ e = [fill\ c\ x' \mid (c,x) \leftarrow split\ e, x' \leftarrow reduce\ x]$

In particular, the function *reduce* implements the basic rule for addition, while *trans* implements the contextual rule by first splitting the given expression into all possible context and expression pairs, then considering any reduction that can made by each component expression, and finally, filling the resulting expressions back into the context.

As with the original small-step semantics in the previous section, the above semantics for expressions does not specify an evaluation order for addition, and is hence non-deterministic. However, if we do wish to specify a particular order, it is straightforward to modify the language of contexts to achieve this. For example, modifying the second case for addition as shown below (and adapting the notion of hole filling accordingly), would ensure the first argument to addition is evaluated before the second.

$$C \ ::= \ [-] \mid C + E \mid \mathbb{Z} + C$$

We'll see another approach to specifying evaluation order in Section 8, when we consider the idea of transforming semantics into abstract machines.

Further reading Contexts are related to a number of other important concepts in programming and semantics, including the use of continuations to make control flow explicit (Reynolds, 1972), navigating around data structures using zippers (Huet, 1997), implementing languages using abstract machines (Ager *et al.*, 2003a), and the idea of differentiating (Abbott *et al.*, 2005) and dissecting (McBride, 2008) datatypes.

6 Big-Step Semantics

Whereas small-step semantics focus on single execution steps, *big-step* semantics specify how terms can be fully executed in one large step. Formally, a big-step operational semantics, also known as a 'natural semantics' (Kahn, 1987), for a language T of syntactic terms comprises two components: a set V of *values*, and an *evaluation relation* between T and V that relates each term to all values that can be reached by fully executing the term. If a term t and a value v are related, we say that t can evaluate to v, and write this as $t \Downarrow v$.

Arithmetic expressions of type *Expr* have a simple big-step operational semantics, given by taking V as the Haskell type *Integer*, and defining the evaluation relation between *Expr* and *Integer* by the following two inference rules:

$$\frac{}{Val\ n \Downarrow n} \qquad\qquad \frac{x \Downarrow n \qquad y \Downarrow m}{Add\ x\ y \Downarrow n + m}$$

The first rule states that a value evaluates to the underlying integer, and the second that if two expressions x and y evaluate respectively to the integer values n and m, then the addition of these expressions evaluates to the integer $n + m$.

The evaluation relation can be translated into a Haskell function definition in a similar manner to the small-step semantics, by using the comprehension notation to return the list of all values that can be reached by executing a given expression to completion:

$$
\begin{aligned}
&eval :: Expr \to [Integer] \\
&eval\ (Val\ n) \quad\;\; = [n] \\
&eval\ (Add\ x\ y) = [n + m \mid n \leftarrow eval\ x, m \leftarrow eval\ y]
\end{aligned}
$$

For our simple expression language, the big-step semantics above is essentially the same as the denotational semantics we presented earlier, but specified in a relational manner using inference rules rather than a functional manner using equations. However, there is no need for a big-step semantics to be compositional, whereas this is a key aspect of the denotational approach. This difference becomes evident when more sophisticated languages are considered. For example, the lambda calculus compiler in (Bahr & Hutton, 2015) is based on a non-compositional semantics specified in big-step form.

Further reading Big-step operational semantics can be useful in situations when we are only interested in the final result of executing a term. In contrast, the small-step approach can be useful when the fine structure of execution is important, such as when considering concurrent languages (Milner, 1999), abstract machines (Hutton & Wright, 2006) or efficiency (Hope & Hutton, 2006). Further examples of the two approaches can be found in any textbook on semantics, such as (Pierce, 2002; Harper, 2016).

7 Rule Induction

Once a semantics for a language has been defined, it can be used as the basis for proving properties of the language. Given that terms and their semantics are built up inductively, such proofs typically proceed using some form of induction. In the case of denotational semantics, the basic proof technique is the familiar idea of structural induction (Burstall, 1969), which allows us to perform proofs by considering the syntactic structure of terms. For operational semantics, the basic technique is the perhaps less familiar but just as useful concept of *rule induction* (Winskel, 1993), which allows us to perform proofs by considering the structure of the rules that are used to define the semantics.

We introduce the idea of rule induction using a simple syntactic example, and then show how the idea can be used to prove a simple semantic property. We begin by inductively defining a set S of non-empty strings of stars by the following two rules:

$$
\frac{}{\star \in S} \qquad\qquad \frac{s \in S}{s\,s \in S}
$$

The first rule, the base case, states that a single star $\star$ is in the set S. The second rule, the inductive case, states that for any string s in S, the string $s\,s$ formed by concatenating s with itself is also in S. Moreover, the inductive nature of the definition means that there is nothing in S beyond those strings obtained by applying these rules a finite number of times, which is sometimes called the ‘extremal clause’ of the definition.

Note that unlike sets that correspond to recursive datatypes, such as binary trees of stars built up using a node operator, there is not a unique way to decompose each element of S in terms of $\star$ and concatenation. For example, the string $\star\star\star$ could be decomposed as $(\star\star)\star$ or $\star(\star\star)$, because concatenation is associative. In general, all that we know is that $\star$ and concatenation are jointly surjective, i.e. all elements of S can be generated using them. This is a key difference between rule induction and structural induction.

For the inductively defined set S, the principle of rule induction states that in order to prove that some property P holds for all elements of S, it suffices to show that P holds for a single star $\star$, the base case, and that if P holds for any element s in S then it also holds for $s\,s$, the inductive case. That is, we have the following proof rule:

$$\frac{P\,(\star) \qquad \forall s \in S.\, P\,(s) \;\Rightarrow\; P\,(s\,s)}{\forall s \in S.\, P\,(s)}$$

This basic scheme can easily be generalised to multiple base and inductive cases, to rules with multiple preconditions, and so on. For example, in the case of our small-step semantics for expressions, we have one base case and two inductive cases:

$$\frac{}{\mathit{Add}\;(\mathit{Val}\;n)\;(\mathit{Val}\;m) \;\longrightarrow\; \mathit{Val}\;(n+m)}$$

$$\frac{x \;\longrightarrow\; x'}{\mathit{Add}\;x\;y \;\longrightarrow\; \mathit{Add}\;x'\;y} \qquad\qquad \frac{y \;\longrightarrow\; y'}{\mathit{Add}\;x\;y \;\longrightarrow\; \mathit{Add}\;x\;y'}$$

Hence, if we want to show that some property $P\,(x,x')$ holds for all transitions $x \;\longrightarrow\; x'$, we can use the principle of rule induction, which in this case has the following form:

$$\frac{\begin{array}{c} P\,(\mathit{Add}\;(\mathit{Val}\;n)\;(\mathit{Val}\;m), \mathit{Val}\;(n+m)) \\ \forall x \longrightarrow x'.\, P\,(x,x') \;\Rightarrow\; P\,(\mathit{Add}\;x\;y, \mathit{Add}\;x'\;y) \\ \forall y \longrightarrow y'.\, P\,(y,y') \;\Rightarrow\; P\,(\mathit{Add}\;x\;y, \mathit{Add}\;x\;y') \end{array}}{\forall x \longrightarrow x'.\, P\,(x,x')}$$

That is, we must show that P holds for the transition defined by the base rule of the semantics, that if P holds for the precondition transition for the first inductive rule then it also holds for the resulting transition, and similarly for the second inductive rule. Note that the three premises are presented vertically in the above rule for reasons of space, and we write $\forall x \longrightarrow y.\, P\,(x,y)$ as shorthand for $\forall x,y.\; x \longrightarrow y \;\Rightarrow\; P\,(x,y)$.

By way of example, we can use rule induction to verify a simple relationship between our small-step and denotational semantics for expressions, namely that making a transition does not change the denotation of an expression:

$$\forall x \longrightarrow x'.\; [\![x]\!] = [\![x']\!]$$

In order to prove this result, we first define the underlying predicate P, then apply rule induction, and finally expand out the definition of P to leave three conditions:

$$\begin{array}{ll}
 & \forall x \;\longrightarrow\; x'.\; [\![x]\!] = [\![x']\!] \\
\Leftrightarrow & \{\text{ define } P\,(x,x') \;\Leftrightarrow\; [\![x]\!] = [\![x']\!] \;\} \\
 & \forall x \;\longrightarrow\; x'.\; P\,(x,x') \\
\Leftarrow & \{\text{ rule induction for } \longrightarrow \} \\
 & P\,(\mathit{Add}\;(\mathit{Val}\;n)\;(\mathit{Val}\;m), \mathit{Val}\;(n+m)) \\
 & \forall x \;\longrightarrow\; x'.\; P\,(x,x') \;\Rightarrow\; P\,(\mathit{Add}\;x\;y, \mathit{Add}\;x'\;y) \\
 & \forall y \;\longrightarrow\; y'.\; P\,(y,y') \;\Rightarrow\; P\,(\mathit{Add}\;x\;y, \mathit{Add}\;x\;y') \\
\Leftrightarrow & \{\text{ definition of } P \;\}
\end{array}$$

$$\begin{array}{l}
[\![Add\ (Val\ n)\ (Val\ m)]\!] = [\![Val\ (n+m)]\!] \\
\forall x \longrightarrow x'\ .\ [\![x]\!] = [\![x']\!] \ \Rightarrow\ [\![Add\ x\ y]\!] = [\![Add\ x'\ y]\!] \\
\forall y \longrightarrow y'\ .\ [\![y]\!] = [\![y']\!] \ \Rightarrow\ [\![Add\ x\ y]\!] = [\![Add\ x\ y']\!]
\end{array}$$

The three final conditions can then be verified by simple calculations over the denotational semantics for expressions, which we include below for completeness:

$$\begin{array}{cl}
 & [\![Add\ (Val\ n)\ (Val\ m)]\!] \\
= & \{ \text{ definition of } [\![\ ]\!] \ \} \\
 & [\![Val\ n]\!] + [\![Val\ m]\!] \\
= & \{ \text{ definition of } [\![\ ]\!] \ \} \\
 & n+m \\
= & \{ \text{ definition of } [\![\ ]\!] \ \} \\
 & [\![Val\ (n+m)]\!]
\end{array}$$

and

$$\begin{array}{cl}
 & [\![Add\ x\ y]\!] \\
= & \{ \text{ definition of } [\![\ ]\!] \ \} \\
 & [\![x]\!] + [\![y]\!] \\
= & \{ \text{ assumption that } [\![x]\!] = [\![x']\!] \ \} \\
 & [\![x']\!] + [\![y]\!] \\
= & \{ \text{ definition of } [\![\ ]\!] \ \} \\
 & [\![Add\ x'\ y]\!]
\end{array}$$

and

$$\begin{array}{cl}
 & [\![Add\ x\ y]\!] \\
= & \{ \text{ definition of } [\![\ ]\!] \ \} \\
 & [\![x]\!] + [\![y]\!] \\
= & \{ \text{ assumption that } [\![y]\!] = [\![y']\!] \ \} \\
 & [\![x]\!] + [\![y']\!] \\
= & \{ \text{ definition of } [\![\ ]\!] \ \} \\
 & [\![Add\ x\ y']\!]
\end{array}$$

We conclude with two further remarks. First of all, this result can also be proved using structural induction, but the proof is simpler and more direct using rule induction. In particular, using structural induction, in the base case for *Val n* we would need to argue that the result is trivially true because there is no transition rule for values, while in the inductive case for *Add x y* we would need to perform a further case analysis depending on which of the three inference rules for addition is applicable. In contrast, using rule induction the proof proceeds directly on the structure of the transition rules, which is the key structure here and gives a proof with three cases, rather than the syntactic structure of expressions, which is secondary and results in a proof with two extra cases.

Secondly, just as proofs using structural induction do not normally proceed in full detail by explicitly defining a predicate and stating the induction principle being used, so the same is true with rule induction. For example, the above proof would often be abbreviated by simply stating that it proceeds by rule induction on the transition $x \longrightarrow x'$, and then immediately stating and verifying the three conditions as above.

Further reading Wright (2005) demonstrates how the principle of rule induction can be used to verify the equivalence of small and big-step operational semantics for our simple expression language. The same idea can also be applied to more general languages, such as versions of the lambda calculus that count evaluation steps (Hope, 2008) or support a form of non-deterministic choice (Moran, 1998).

8 Abstract Machines

All of the examples we have considered so far have been focused on explaining semantic ideas. In this section, we show how the language of integers and addition can also be used to help discover semantic ideas. In particular, we show how it can be used as the basis for discovering how to implement an *abstract machine* (Landin, 1964) for evaluating expressions in a manner that precisely defines the order of evaluation.

We begin by recalling the following simple evaluation function from Section 3:

```
eval :: Expr → Integer
eval (Val n)   = n
eval (Add x y) = eval x + eval y
```

As noted previously, this definition does not specify the order in which the two arguments of addition are evaluated. Rather, this is determined by the implementation of the meta-language, in this case Haskell. If desired, the order of evaluation can be made explicit by defining an abstract machine for evaluating expressions, which uses a 'control stack' to specify how the machine should behave after evaluating the current expression. In other words, the control stack is used to keep track of what should be done next.

Formally, an abstract machine is usually defined by a set of syntactic rewrite rules that make explicit how each step of evaluation proceeds. In Haskell, this idea can be realised by mutually defining a set of first-order, tail recursive functions on suitable data structures. In our case, an abstract machine for evaluating expressions of type *Expr* can be defined by three components: a type *Cont* of control stacks, a function $eval' :: Expr \to Cont \to Integer$ that evaluates an expression and then continues by executing the given control stack, and finally, a function $exec :: Cont \to Integer \to Integer$ that executes a control stack given the integer that resulted from evaluating an expression. The desired relationship between the component functions is captured by the following simple equation:

$$eval'\ e\ c \quad = \quad exec\ c\ (eval\ e) \tag{1}$$

That is, evaluating an expression and then executing a control stack should give the same result as executing the control stack using the value of the expression.

At this point in most presentations, definitions for *Cont*, *eval'* and *exec* would now be given, from which the above equation could then be proved. However, we can also view the equation as a *specification* for these three components, from which we then aim to discover, or *calculate* definitions that satisfy the specification. Given that the specification has two knowns (*Expr* and *eval*) and three unknowns (*Cont*, *eval'* and *exec*), this may seem like an impossible task. However, with the benefit of experience gained from studying the simple expression language for many years, it turns out to be straightforward. Note that the above specification has many possible solutions. For example, the machine could be implemented using different forms of control stacks, or adopt different evaluation orders. We develop one possible solution below, but others are possible too.

To calculate an abstract machine we proceed from specification (1) by structural induction on the expression e. In each case, we start with the right-hand side $exec\ c\ (eval\ e)$ of the specification and gradually transform it by equational reasoning, aiming to end up with a term t that does not refer to the original evaluation function $eval$, such that we can then take $eval'\ e\ c = t$ as a defining equation for $eval'$ in this case. In order to do this we will find that we need to introduce new constructors into the control stack type *Cont*, along with their interpretation by the execution function *exec*.

For the base case, $e = Val\ n$, the calculation has just one step:

$$\begin{aligned} & exec\ c\ (eval\ (Val\ n)) \\ = & \quad \{\ \text{applying } eval\ \} \\ & exec\ c\ n \end{aligned}$$

The resulting term $exec\ c\ n$ already has the required form (does not refer to *eval*), from which we conclude that the following definition satisfies (1) in the base case:

$$eval'\ (Val\ n)\ c = exec\ c\ n$$

That is, if the expression is an integer value it is already fully evaluated, and we simply execute the control stack using this integer as an argument.

For the inductive case, $e = Add\ x\ y$, we begin in the same way as above:

$$\begin{aligned} & exec\ c\ (eval\ (Add\ x\ y)) \\ = & \quad \{\ \text{applying } eval\ \} \\ & exec\ c\ (eval\ x + eval\ y) \end{aligned}$$

No further definitions can be applied at this point. However, as we are performing an inductive calculation, we can make use of the induction hypotheses for the expressions x and y. In order to use the induction hypothesis for y, which is $eval'\ y\ c' = exec\ c'\ (eval\ y)$, we must rewrite the term $exec\ c\ (eval\ x + eval\ y)$ that is being manipulated into the form $exec\ c'\ (eval\ y)$ for some control stack c'. That is, we need to solve the equation:

$$exec\ c'\ (eval\ y) \quad = \quad exec\ c\ (eval\ x + eval\ y)$$

First of all, we generalise from the specific values $eval\ x$ and $eval\ y$ to give:

$$exec\ c'\ m \quad = \quad exec\ c\ (n + m)$$

Note that we can't simply use this equation as a definition for *exec*, because the integer n and control stack c would be unbound in the body of the definition as they do not appear on the left-hand side. The solution is to package these two variables up in the control stack argument c' (which can freely be instantiated as it is existentially quantified) by adding a new constructor to the *Cont* type that takes these two variables as arguments,

$$ADD :: Integer \to Cont \to Cont$$

and defining a new equation for *exec* as follows:

$$exec\ (ADD\ n\ c)\ m = exec\ c\ (n + m)$$

That is, executing a control stack of the form *ADD n c* given an integer argument *m* proceeds by simply adding the two integers *n* and *m* and then executing the remaining control stack *c*, hence the choice of the name for the new constructor.

Using the above ideas, we now continue the calculation:

$$\begin{array}{ll}
& exec\ c\ (eval\ x + eval\ y) \\
= & \{\ \text{define: } exec\ (ADD\ n\ c)\ m = exec\ c\ (n+m)\ \} \\
& exec\ (ADD\ (eval\ x)\ c)\ (eval\ y) \\
= & \{\ \text{induction hypothesis for } y\ \} \\
& eval'\ y\ (ADD\ (eval\ x)\ c)
\end{array}$$

No further definitions can now be applied, so we seek to use the induction hypothesis for x, which is $eval'\ x\ c' = exec\ c'\ (eval\ x)$. In order to use this, we must rewrite the term $eval'\ y\ (ADD\ (eval\ x)\ c)$ that is being manipulated into the form $exec\ c'\ (eval\ x)$ for some control stack c'. That is, we need to solve the following equation:

$$exec\ c'\ (eval\ x) \quad = \quad eval'\ y\ (ADD\ (eval\ x)\ c)$$

As with the case for y, we first generalise from $eval\ x$ to give

$$exec\ c'\ n \quad = \quad eval'\ y\ (ADD\ n\ c)$$

and then package up the free variables y and c into the argument c' by adding a new constructor to *Cont* that takes these variables as arguments

$$EVAL :: Expr \to Cont \to Cont$$

and defining a new equation for *exec* as follows:

$$exec\ (EVAL\ y\ c)\ n = eval'\ y\ (ADD\ n\ c)$$

That is, executing a control stack of the form *EVAL y c* given an integer argument *n* proceeds by evaluating the expression *y* and then executing the control stack *ADD n c*. Using these ideas, the calculation can now be completed:

$$\begin{array}{ll}
& eval'\ y\ (ADD\ (eval\ x)\ c) \\
= & \{\ \text{define: } exec\ (EVAL\ y\ c)\ n = eval'\ y\ (ADD\ n\ c)\ \} \\
& exec\ (EVAL\ y\ c)\ (eval\ x) \\
= & \{\ \text{induction hypothesis for } x\ \} \\
& eval'\ x\ (EVAL\ y\ c)
\end{array}$$

The final term now has the required form (does not refer to *eval*), from which we conclude that the following definition satisfies specification (1) in the inductive case:

$$eval'\ (Add\ x\ y)\ c = eval'\ x\ (EVAL\ y\ c)$$

That is, if the expression is an addition we proceed by evaluating the first argument expression *x*, with the term *EVAL y* placed on the control stack to indicate that the second argument expression *y* should be evaluated once that of *x* has completed. In this manner, the definition makes explicit that addition is evaluated in left-to-right order. We could equally well have chosen the opposite order, by simply reversing the order in which we apply the induction hypotheses for x and y. We have this freedom in the calculation because the semantics defined by *eval* does not specify an evaluation order.

Finally, we conclude the development of the abstract machine by redefining the original evaluation function $\mathit{eval} :: \mathit{Expr} \to \mathit{Integer}$ in terms of the new function $\mathit{eval}' :: \mathit{Expr} \to \mathit{Cont} \to \mathit{Integer}$. In this case there is no need to use induction as simple calculation suffices, during which we introduce a new constructor $\mathit{HALT} :: \mathit{Cont}$ to transform the term being manipulated into the required form in order that specification (1) can be applied:

$$\begin{aligned}
& \mathit{eval}\ e \\
= \quad & \{ \text{ define: } \mathit{exec}\ \mathit{HALT}\ n = n \ \} \\
& \mathit{exec}\ \mathit{HALT}\ (\mathit{eval}\ e) \\
= \quad & \{ \text{ specification (1) } \} \\
& \mathit{eval}'\ e\ \mathit{HALT}
\end{aligned}$$

In conclusion, we have calculated the following definitions, which together implement an abstract machine for evaluating simple arithmetic expressions:

$$\begin{array}{l}
\textbf{data}\ \mathit{Cont} = \mathit{HALT} \mid \mathit{EVAL}\ \mathit{Expr}\ \mathit{Cont} \mid \mathit{ADD}\ \mathit{Integer}\ \mathit{Cont} \\[1ex]
\mathit{eval} :: \mathit{Expr} \to \mathit{Integer} \\
\mathit{eval}\ e = \mathit{eval}'\ e\ \mathit{HALT} \\[1ex]
\mathit{eval}' :: \mathit{Expr} \to \mathit{Cont} \to \mathit{Integer} \\
\mathit{eval}'\ (\mathit{Val}\ n) \qquad c \ = \mathit{exec}\ c\ n \\
\mathit{eval}'\ (\mathit{Add}\ x\ y) \quad c \ = \mathit{eval}'\ x\ (\mathit{EVAL}\ y\ c) \\[1ex]
\mathit{exec} :: \mathit{Cont} \to \mathit{Integer} \to \mathit{Integer} \\
\mathit{exec}\ \mathit{HALT} \qquad\quad n \ = n \\
\mathit{exec}\ (\mathit{EVAL}\ y\ c)\ n \ = \mathit{eval}'\ y\ (\mathit{ADD}\ n\ c) \\
\mathit{exec}\ (\mathit{ADD}\ n\ c)\ \ m = \mathit{exec}\ c\ (n+m)
\end{array}$$

Note that *eval′* and *exec* are mutually recursive, which corresponds to the machine having two modes of operation, depending on whether it is currently being driven by the structure of the expression or the control stack. For example, for $1+2$ we have:

$$\begin{aligned}
& \mathit{eval}\ (\mathit{Add}\ (\mathit{Val}\ 1)\ (\mathit{Val}\ 2)) \\
= \quad & \mathit{eval}'\ (\mathit{Add}\ (\mathit{Val}\ 1)\ (\mathit{Val}\ 2))\ \mathit{HALT} \\
= \quad & \mathit{eval}'\ (\mathit{Val}\ 1)\ (\mathit{EVAL}\ (\mathit{Val}\ 2)\ \mathit{HALT}) \\
= \quad & \mathit{exec}\ (\mathit{EVAL}\ (\mathit{Val}\ 2)\ \mathit{HALT})\ 1 \\
= \quad & \mathit{eval}'\ (\mathit{Val}\ 2)\ (\mathit{ADD}\ 1\ \mathit{HALT}) \\
= \quad & \mathit{exec}\ (\mathit{ADD}\ 1\ \mathit{HALT})\ 2 \\
= \quad & \mathit{exec}\ \mathit{HALT}\ 3 \\
= \quad & 3
\end{aligned}$$

In summary, we have shown how to calculate an abstract machine for evaluating arithmetic expressions, with all of the implementation machinery falling naturally out of the calculation process. In particular, we required no prior knowledge of the implementation ideas, as these were systematically discovered during the calculation. Moreover, our approach only required elementary equational reasoning techniques, and avoided the need for more sophisticated concepts such as continuations and defunctionalisation that are traditionally used in the derivation of abstract machines. Focusing on the simple language of integers and addition was key to our discovery of this simpler approach.

Further reading Reynold's seminal paper (1972) introduced the three key techniques that have traditionally been used to derive implementations from semantics: definitional interpreters, continuation-passing style, and defunctionalisation. Using these techniques, Danvy and his collaborators showed how semantics could be systematically transformed into abstract machines and compilers (Ager *et al.*, 2003a; Ager *et al.*, 2003b; Danvy & Nielsen, 2004; Danvy, 2008; Danvy & Millikin, 2009). Using the idea of dissecting a datatype, McBride (2008) developed a generic recipe that turns a denotational semantics expressed using a fold operator into an equivalent abstract machine. A survey of abstract machines for different kinds of languages is given in (Diehl *et al.*, 2000).

This section is based upon (Hutton & Wright, 2006; Hutton & Bahr, 2016), which also show how to calculate machines for extended versions of our expression language. Similar techniques can be used to calculate compilers for stack (Bahr & Hutton, 2015) and register machines (Hutton & Bahr, 2017; Bahr & Hutton, 2020), typed languages (Pickard & Hutton, 2021) and non-terminating languages (Bahr & Hutton, 2022).

9 Summary and Conclusion

In this article we have show how a range of semantic concepts can be presented in a simple manner using the language of integers and addition. We have considered various semantic approaches, how induction principles can be used to reason about semantics, and how semantics can be transformed into implementations. In each case, using a minimal language allowed us to present the ideas in a clear and concise manner, by avoiding the additional complexity that comes from considering more sophisticated languages.

Of course, using a simple language also has limitations. For example, it may not be sufficient to illustrate the differences between semantic approaches. As a case in point, when we presented the big-step semantics for arithmetic expressions, we found that it was essentially the same as the denotational semantics, except that it was formulated using inference rules rather than equations. Moreover, a simple language by its very nature does not raise semantic questions and challenges that arise with more complex languages. For example, features such as mutable state, variable binding, and concurrency are particularly interesting from a semantic point of view, especially when used in combination.

For readers interested in learning more about semantics, there are many excellent textbooks such as (Harper, 2016; Pierce, 2002; Reynolds, 1998; Winskel, 1993), summer schools including the Oregon Programming Languages Summer School (OPLSS, 2022) and the Midlands Graduate School (MGS, 2022), and numerous online resources. We hope that our simple language provides others with a useful gateway and tool for exploring further aspects of programming language semantics. In this setting, it's easy as 1,2,3.

Acknowledgements

I would like to thank Jeremy Gibbons, Ralf Hinze, Peter Thiemann, Andrew Tolmach and the anonymous reviewers for many useful comments and suggestions that significantly improved the article. This work was funded by EPSRC grant EP/P00587X/1, *Unified Reasoning About Program Correctness and Efficiency*.

Conflicts of Interest

None.

References

Abbott, Michael Gordon, Altenkirch, Thorsten, McBride, Conor, & Ghani, Neil. (2005). δ for Data: Differentiating Data Structures. *Fundamenta Informaticae*, **65**(1-2).

Abramsky, Samson, & Jung, Achim. (1994). Domain Theory. *Handbook of Logic in Computer Science*, vol. 3. Clarendon Press.

Abramsky, Samson, & McCusker, Guy. (1999). Game Semantics. *Computational Logic*.

Ager, Mads Sig, Biernacki, Dariusz, Danvy, Olivier, & Midtgaard, Jan. (2003a). A Functional Correspondence Between Evaluators and Abstract Machines. *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*.

Ager, Mads Sig, Biernacki, Dariusz, Danvy, Olivier, & Midtgaard, Jan. (2003b). *From Interpreter to Compiler and Virtual Machine: A Functional Derivation*. Research Report RS-03-14. BRICS, Department of Computer Science, University of Aarhus.

Bahr, Patrick, & Hutton, Graham. (2015). Calculating Correct Compilers. *Journal of Functional Programming*, **25**.

Bahr, Patrick, & Hutton, Graham. (2020). Calculating Correct Compilers II: Return of the Register Machines. *Journal of Functional Programming*, **30**.

Bahr, Patrick, & Hutton, Graham. (2022). Monadic Compiler Calculation. *Proceedings of the ACM on Programming Languages*, **6**(ICFP).

Burstall, Rod. (1969). Proving Properties of Programs by Structural Induction. *The Computer Journal*, **12**(1).

Danvy, Olivier. (2008). Defunctionalized Interpreters for Programming Languages. *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*.

Danvy, Olivier, & Millikin, Kevin. (2009). Refunctionalization at Work. *Science of Computer Programming*, **74**(8).

Danvy, Olivier, & Nielsen, Lasse R. (2004). *Refocusing in Reduction Semantics*. Research Report RS-04-26. BRICS, Department of Computer Science, University of Aarhus.

Diehl, Stephan, Hartel, Pieter, & Sestoft, Peter. (2000). Abstract Machines for Programming Language Implementation. *Future Generation Computer Systems*, **16**(05).

Duignan, Brian. (2018). *Occam's Razor*. Encyclopedia Britannica. Available online from `https://www.britannica.com/topic/Occams-razor`.

Dybjer, Peter. (1994). Inductive Families. *Formal Aspects of Computing*, **6**(4).

Felleisen, Matthias, & Hieb, Robert. (1992). The Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science*, **103**(2).

Gibbons, Jeremy, & Jones, Geraint. (1998). The Under-Appreciated Unfold. *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*.

Goguen, Joseph, & Malcolm, Grant. (1996). *Algebraic Semantics of Imperative Programs*. MIT Press.

Harper, Robert. (2016). *Practical Foundations for Programming Languages (2nd edition)*. Cambridge University Press.

Hoare, Tony. (1969). An Axiomatic Basis for Computer Programming. *Communications of the ACM*, **12**.

Hope, Catherine. (2008). *A Functional Semantics for Space and Time*. Ph.D. thesis, University of Nottingham.

Hope, Catherine, & Hutton, Graham. (2006). Accurate Step Counting. *Implementation and Application of Functional Languages*. LNCS, vol. 4015. Springer Berlin / Heidelberg.

Hu, Liyang, & Hutton, Graham. (2009). Towards a Verified Implementation of Software Transactional Memory. *Trends in Functional Programming Volume 9*. Intellect.

Hu, Liyang, & Hutton, Graham. (2010). Compiling Concurrency Correctly: Cutting Out The Middle Man. *Trends in Functional Programming Volume 10*. Intellect.

Huet, Gerard. (1997). The Zipper. *Journal of Functional Programming*, **7**(5).

Hutton, Graham. (1998). Fold and Unfold for Program Semantics. *Proceedings of the 3rd International Conference on Functional Programming*.

Hutton, Graham, & Bahr, Patrick. (2016). Cutting Out Continuations. *A List of Successes That Can Change the World*. LNCS, vol. 9600. Springer.

Hutton, Graham, & Bahr, Patrick. (2017). Compiling a 50-Year Journey. *Journal of Functional Programming*, **27**.

Hutton, Graham, & Wright, Joel. (2004). Compiling Exceptions Correctly. *Proceedings of the 7th International Conference on Mathematics of Program Construction*. LNCS, vol. 3125. Springer.

Hutton, Graham, & Wright, Joel. (2006). Calculating an Exceptional Machine. *Trends in Functional Programming Volume 5*. Intellect.

Hutton, Graham, & Wright, Joel. (2007). What is the Meaning of These Constant Interruptions? *Journal of Functional Programming*, **17**(6).

Kahn, Gilles. (1987). Natural Semantics. *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*.

Landin, Peter. (1964). The Mechanical Evaluation of Expressions. *The Computer Journal*, **6**.

McBride, Conor. (2008). Clowns to the Left of Me, Jokers to the Right: Dissecting Data Structures. *Proceedings of the Symposium on Principles of Programming Languages*.

McCarthy, John, & Painter, James. (1967). Correctness of a Compiler for Arithmetic Expressions. *Pages 33–41 of: Mathematical Aspects of Computer Science*. Proceedings of Symposia in Applied Mathematics, vol. 19. American Mathematical Society.

Meijer, Erik, Fokkinga, Maarten, & Paterson, Ross. (1991). Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. *Proceedings of the Conference on Functional Programming and Computer Architecture*.

MGS. (2022). *Midlands Graduate School in the Foundations of Computing Science*. `http://www.cs.nott.ac.uk/MGS/`.

Milner, Robin. (1999). *Communicating and Mobile Systems: the Pi Calculus*. Cambridge University Press.

Moran, Andrew. 1998 (Sept.). *Call-By-Name, Call-By-Need, and McCarthy's Amb*. Ph.D. thesis, Chalmers University of Technology.

Mosses, Peter. (2005). *Action Semantics*. Cambridge University Press.

Norell, Ulf. (2007). *Towards a Practical Programming Language Based on Dependent Type Theory*. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology.

OPLSS. (2022). *Oregon Programming Languages Summer School*. `https://www.cs.uoregon.edu/research/summerschool/archives.html`.

Pickard, Mitchell, & Hutton, Graham. (2021). Calculating Dependently-Typed Compilers. *Proceedings of the ACM on Programming Languages*, **5**(ICFP).

Pierce, Benjamin. (2002). *Types and Programming Languages*. MIT Press.

Plotkin, Gordon. (1981). *A Structured Approach to Operational Semantics*. Report DAIMI-FN-19. Computer Science Department, Aarhus University, Denmark.

Plotkin, Gordon. (2004). The Origins of Structural Operational Semantics. *The Journal of Logic and Algebraic Programming*, **60-61**.

Reynolds, John C. (1972). Definitional Interpreters for Higher-Order Programming Languages. *Proceedings of the ACM Annual Conference*.

Reynolds, John C. (1998). *Theories of Programming Languages*. Cambridge University Press.

Schmidt, David A. (1986). *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc.

Scott, Dana, & Strachey, Christopher. (1971). *Toward a Mathematical Semantics for Computer Languages*. Technical Monograph PRG-6. Oxford Programming Research Group.

Wadler, Philip. (1998). *The Expression Problem*. Available online from `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`.

Wand, Mitchell. (1982). Deriving Target Code as a Representation of Continuation Semantics. *ACM Transactions on Programming Languages and Systems*, **4**(3).

Winskel, Glynn. (1993). *The Formal Semantics of Programming Languages: An Introduction*. MIT Press.

Wright, Joel. (2005). *Compiling and Reasoning about Exceptions and Interrupts*. Ph.D. thesis, University of Nottingham.