*JFP* **33**, e9, 26 pages, 2023. © The Author(s), 2023. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution and reproduction, provided the original article is properly cited.
doi:10.1017/S0956796823000072

# *Programming language semantics: It's easy as 1,2,3*

GRAHAM HUTTON
*University of Nottingham, Nottingham NG7 2RD, UK*
(*e-mail:* graham.hutton@nottingham.ac.uk)

## Abstract

Programming language semantics is an important topic in 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 is easy as 1,2,3.

## 1 Introduction

Semantics is the general term for the study of meaning. In computer science, the subject of *programming language semantics* seeks to give precise mathematical meaning to programs. When studying a new subject, it can be 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 & Painter (1967), Wand (1982) and Wadler (1998), to name but a few. However, this article is the first to use the language as a general tool for exploring a range of different semantics 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 and variable capture. 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 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 the author's own work.

The article written in a tutorial style 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 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) and illustrate how inductive techniques can be used to reason about semantics. In turn, those with more experience may wish to proceed quickly through to section 8, which presents an extended example of how abstract machines can be systematically derived from semantics using the concepts of continuations and defunctionalisation.

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 the minimal approach, and provide pointers to further reading. Haskell is used throughout as a meta-language 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 the 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 supports the use of simple equational reasoning techniques for proving properties of 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 a valuation 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 $y$.

Compositionality simplifies reasoning because it allows us to replace 'equals by equals'. For example, our expression semantics satisfies the following property:

$$\frac{[\![x]\!] = [\![x']\!] \qquad [\![y]\!] = [\![y']\!]}{[\![Add\ x\ y]\!] = [\![Add\ x'\ y']\!]}$$

That is, we can freely replace the two argument expressions of an addition by other expressions with the same meanings, without changing the meaning of the addition as a whole. This property can be proved by simple equational reasoning using the definition of the valuation function and the assumptions about the argument expressions:

$$\begin{array}{ll} & [\![Add\ x\ y]\!] \\ = & \{\ \text{definition of } [\![-]\!]\ \} \\ & [\![x]\!] + [\![y]\!] \\ = & \{\ \text{assumptions}\ \} \\ & [\![x']\!] + [\![y']\!] \\ = & \{\ \text{definition of } [\![-]\!]\ \} \\ & [\![Add\ x'\ y']\!] \end{array}$$

In practice, given that terms and their semantics are built up inductively, proofs about denotational semantics typically proceed using structural induction (Burstall, 1969). By way of example, let us show that our expression semantics is *total*, in the sense that for every expression $e$ there is an integer $n$ such that $[\![e]\!] = n$.

The proof of totality proceeds by induction on the structure of the expression $e$. For the base case, $e = Val\,n$, the equation $[\![Val\,n]\!] = n$ is trivially true by the definition of the valuation function. For the inductive case, $e = Add\,x\,y$, we can assume by induction that $[\![x]\!] = n$ and $[\![y]\!] = m$ for some integers $n$ and $m$, and it then follows using the valuation function that $[\![Add\,x\,y]\!] = [\![x]\!] + [\![y]\!] = n + m$, which establishes this case is also true.

The valuation 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:

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

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 $g$:

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

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:

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

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 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 make evaluation order explicit this requires the introduction of additional structure into the semantics, which we will discuss when we consider abstract machines in Section 8.

**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), nondeterminism (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 7.

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.

For example, 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 to give a single value and is called a *reduction* (or contraction) rule as it specifies how a basic operation is performed. An expression that matches such a rule is termed a *reducible expression* or ‘redex’. In turn, the last two rules permit transitions to be made on either side of an addition and are known as *structural* (or congruence) rules, as they specify how larger terms can be reduced.

Note that the semantics is non-deterministic, because an expression may 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 reduction rule can be applied on either side of the top-level addition using the two structural rules:

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

Such transitions change the syntactic form of an expression, but the underlying value of the expression remains the same, in this case 10. More formally, we can now capture a simple relationship between our denotational and small-step semantics for expressions, namely that making a transition does not change the denotation of an expression:

$$\frac{e \longrightarrow e'}{[\![e]\!] \;=\; [\![e']\!]}$$

This property can be proved by induction on the structure of the expression $e$. For the base case, $e = \mathit{Val}\,n$, the result is trivially true because there is no transition rule for values in our small-step semantics, and hence the precondition $e \longrightarrow e'$ cannot be satisfied. For the inductive case, $e = \mathit{Add}\,x\,y$, we proceed by performing a further case analysis, depending on which of the three inference rules for addition is applicable:

- If the first inference rule is applicable, i.e. the precondition $e \longrightarrow e'$ has the form $\mathit{Add}\,(\mathit{Val}\,n)\,(\mathit{Val}\,m) \longrightarrow \mathit{Val}\,(n+m)$, then the conclusion $[\![\mathit{Add}\,(\mathit{Val}\,n)\,(\mathit{Val}\,m)]\!] = [\![\mathit{Val}\,(n+m)]\!]$ is true because both sides evaluate to $n+m$.

- If the second rule is applicable, i.e. the precondition $e \longrightarrow e'$ has the form $\mathit{Add}\,x\,y \longrightarrow \mathit{Add}\,x'\,y$ for some transition $x \longrightarrow x'$, then we must show $[\![\mathit{Add}\,x\,y]\!] = [\![\mathit{Add}\,x'\,y]\!]$. Using the ‘equals by equals’ property of addition from the previous section, this equation holds if $[\![x]\!] = [\![x']\!]$ and $[\![y]\!] = [\![y]\!]$, the first of which is true by induction based on the assumption $x \longrightarrow x'$, and the second of which is true by reflexivity.

- If the third rule is applicable, the same form of reasoning as the second case can be used, except that the expression $y$ makes a transition rather than $x$.

While the above proof is correct, it is rather cumbersome, as it involves quite a bit of case analysis. In the next section, we will see how to prove the relationship between the semantics in a simpler and more direct manner, using the principle of rule induction.

Evaluation of an expression using the small-step semantics proceeds by a series of zero or more transition steps. Formally, this is usually captured by taking the reflexive/transitive closure of the transition relation, written as $\overset{*}{\longrightarrow}$. For example, the fact that the expression above evaluates to 10 can be written using this notion as follows:

$$(1+2)+(3+4) \;\overset{*}{\longrightarrow}\; 10$$

By repeated application of the transition relation, we can also 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 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)                 = [ ]
trans (Add (Val n) (Val m)) = [Val (n + m)]
trans (Add x y)               = [Add x′ y | x′ ← trans x] ++ [Add x y′ | y′ ← trans y]
```

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:

```
data Tree a = Node a [Tree a]
exec :: Expr → Tree Expr
exec e = Node e [exec e′ | e′ ← trans e]
```

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:

```
exec :: Expr → Tree Expr
exec = unfold id trans
```

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 is not 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 requires additional structure. Being able to specify evaluation order in a straightforward manner is 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 that are 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 more accessible. 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). 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). 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 Rule induction

For denotational semantics, the basic proof technique is the familiar idea of structural induction, 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 numeric example and then show how it can be used to simplify the semantic proof from the previous section. We begin by inductively defining a set $E$ of even natural numbers by the following two rules:

$$\frac{}{0 \in E} \qquad\qquad \frac{n \in E}{n + 2 \in E}$$

That first rule, the base case, states that the number 0 is the set $E$. The second rule, the inductive case, states that for any number $n$ in $E$, the number $n + 2$ is also in $E$. Moreover, the inductive nature of the definition means there is nothing in the set $E$ beyond the numbers that can be obtained by applying these two rules a finite number of times, which is sometimes called the 'extremal clause' of the definition.

For the inductively defined set $E$, the principle of rule induction states that in order to prove that some property $P$ holds for all elements of $E$, it suffices to show that $P$ holds for 0, the base case, and that if $P$ holds for any element $n \in E$ then it also holds for $n + 2$, the inductive case. That is, we have the following proof rule:

$$\frac{P\,(0) \qquad \forall n \in E.\, P\,(n) \;\Rightarrow\; P\,(n+2)}{\forall n \in E.\, P\,(n)}$$

By way of example, we can use rule induction to verify a simple closure property of even numbers, namely that the addition of two even numbers is also even:

$$\forall n \in E.\, n + n \in E$$

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

$$\begin{array}{ll}
& \forall n \in E.\;\, n + n \in E \\
\Leftrightarrow & \{ \text{ define } P\,(n) \;\Leftrightarrow\; n + n \in E \;\} \\
& \forall n \in E.\;\, P\,(n) \\
\Leftarrow & \{ \text{ rule induction } \} \\
& P\,(0) \;\wedge\; \forall n \in E.\;\, P\,(n) \Rightarrow P\,(n+2) \\
\Leftrightarrow & \{ \text{ definition of } P \;\} \\
& 0 + 0 \in E \;\wedge\; \forall n \in E.\;\, n + n \in E \;\Rightarrow\; (n+2) + (n+2) \in E
\end{array}$$

The first resulting condition simplifies to $0 \in E$, which is trivially true by the first rule that defines the set $E$. In turn, for the second condition the concluding term can be rearranged to $((n + n) + 2) + 2 \in E$, which by applying the second rule for $E$ twice follows from $n + n \in E$, which is true by assumption. Note that this closure property cannot be proved using normal mathematical induction on the natural numbers, because the property is only true for even numbers rather than for arbitrary naturals.

The concept of rule induction can easily be generalised to multiple base and inductive cases, to rules with multiple preconditions and so on. For example, for our small-step semantics of 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 $P\,(e, e')$ on pairs of expressions holds for all transitions $e \longrightarrow e'$, we can use rule induction, which in this case has the form:

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

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 $\forall x, y.\; x \longrightarrow y \;\Rightarrow\; P\,(x, y)$.

We can use the above rule induction principle to verify the relationship between the denotational and small-step semantics for expressions from the previous section, which can be expressed using our shorthand notation as follows:

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

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

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

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

$$\begin{array}{ll}
& [\![\mathit{Add}\,(\mathit{Val}\,n)\,(\mathit{Val}\,m)]\!] \\
= & \{\text{ definition of } [\![\text{-}]\!] \;\}
\end{array}$$

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

and

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

and

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

We conclude with two further remarks. First of all, when compared to the original proof by structural induction in Section 3, the above proof by rule induction is simpler and more direct. In particular, using structural induction, in the base case for $\mathit{Val}\ n$ we needed to argue that the result is trivially true because there is no transition rule for values, while in the inductive case for $\mathit{Add}\ x\ y$ we needed 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 property 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 $e \longrightarrow e'$ 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).

## 6 Contextual semantics

The small-step semantics for expressions in Section 4 has one basic reduction 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, also known as a 'reduction 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 in our small-step semantics:

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

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 $[-]+(3+4)$, where 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*

This style of context is known as 'outside-in', as locating the hole involves navigating from the outside of the context inwards. For example, the concept of filling the hole in a context $c$ with an expression $e$, which we write as $c\,[e]$, can be defined as follows:

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

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 will 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:

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

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 reduction rule and a single structural rule. Moreover, if we subsequently wished to extend the language with other features, this usually only requires adding new reduction 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        e = e
fill (AddL c r) e = Add (fill c e) r
fill (AddR l c) e = Add l (fill c e)
```

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 l r → [(AddL c r, x) | (c, x) ← split l] ++ [(AddR l c, x) | (c, x) ← split r]
```

The behaviour of this function 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 *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 → [Expr]
trans e = [fill c x′ | (c, x) ← split e, x′ ← reduce x]
```

In particular, the function *reduce* implements the reduction 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.

We conclude with two further remarks. First of all, although efficiency is not usually a primary concern when defining semantics, the small-step semantics for expressions in both original and contextual form perform rather poorly in terms of the amount of computation they require. In particular, evaluating an expression using these semantics involves a repeated process of finding the next point where a reduction step can be made, performing the reduction, and then filling the resulting expression back into the original term. This is clearly quite an inefficient way to perform evaluation.

And secondly, as with the original small-step semantics in the previous section, the contextual semantics 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.

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

This version of the semantics also satisfies a *unique decomposition* property, namely that any expression $e$ that is not a value can be uniquely decomposed into the form $e = c\,[x]$ for some context $c$ and reducible expression $x$, which makes precise the sense in which there is at most one possible transition for any expression.

The unique decomposition property can be proved by induction on the expression $e$. For the base case, $e = \mathit{Val}\;n$, the property is trivially true as the expression is already a value. For the inductive case, $e = \mathit{Add}\;l\;r$, we construct a unique decomposition $e = c\,[x]$ by case analysis on the form of the two argument expressions $l$ and $r$:

- If $l$ and $r$ are both values, then $c = [-]$ and $x = \mathit{Add}\;l\;r$ is the only possible decomposition of $e = \mathit{Add}\;l\;r$, as both subterms of $e$ are values and hence not reducible.
- If $l$ is an addition, then by induction $l$ can be uniquely decomposed into the form $l = c'\,[x']$ for some context $c'$ and reducible expression $x'$. Then $c = c' + r$ and $x = x'$ is the only possible decomposition of $e = \mathit{Add}\;l\;r$, as the syntax for contexts specifies that we can only decompose $r$ when $l$ is a value, which it is not.
- Finally, if $l$ has the form $\mathit{Val}\;n$ for some integer $n$, and $r$ is an addition, then by induction $r$ can be uniquely decomposed into the form $r = c'\,[x']$ for some context $c'$ and reducible expression $x'$. Then $c = n + c'$ and $x = x'$ is the only possible decomposition of $e = \mathit{Add}\;l\;r$, as $l$ is already a value and hence cannot be decomposed.

We will see another approach to specifying evaluation order in Section 8 when we consider the idea of transforming semantics into abstract machines, which provide a small-step approach to evaluating expressions that is also more efficient.

**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), deriving abstract machines from evaluators (Ager *et al.*, 2003*a*) and the idea of differentiating (Abbott *et al.*, 2005) and dissecting (McBride, 2008) datatypes. We will return to some of these topics later on when we consider abstract machines.

## 7 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:

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

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:

```
eval :: Expr → [Integer]
eval (Val n)   = [n]
eval (Add x y) = [n + m | n ← eval x, m ← eval y]
```

For our simple expression language, the big-step semantics is essentially the same as the denotational semantics from Section 3 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.

Formally, the fact that the denotational and big-step semantics for the expression language are equivalent can be captured by the following property:

$$[\![e]\!] = n \quad \Leftrightarrow \quad e \Downarrow n$$

That is, an expression denotes an integer value precisely when it evaluates to this value. To prove this result, we consider the two directions separately. In the left-to-right direction, the implication $[\![e]\!] = n \ \Rightarrow \ e \Downarrow n$ can first be simplified by substituting the assumption $n = [\![e]\!]$ into the conclusion $e \Downarrow n$ to give $e \Downarrow [\![e]\!]$, which property can then be verified by structural induction on the expression $e$. For the base case, $e = \mathit{Val}\ n$, we have

$$\begin{array}{ll} & \mathit{Val}\ n \Downarrow [\![\mathit{Val}\ n]\!] \\ \Leftrightarrow & \quad \{ \text{ definition of } [\![\text{-}]\!] \ \} \\ & \mathit{Val}\ n \Downarrow n \\ \Leftrightarrow & \quad \{ \text{ first rule for } \Downarrow \ \} \\ & \mathit{True} \end{array}$$

while for the inductive case, $e = \mathit{Add}\ x\ y$, we reason as follows:

$$\begin{array}{ll} & \mathit{Add}\ x\ y \Downarrow [\![\mathit{Add}\ x\ y]\!] \\ \Leftrightarrow & \quad \{ \text{ definition of } [\![\text{-}]\!] \ \} \\ & \mathit{Add}\ x\ y \Downarrow [\![x]\!] + [\![y]\!] \\ \Leftarrow & \quad \{ \text{ second rule for } \Downarrow \ \} \\ & x \Downarrow [\![x]\!] \ \wedge \ y \Downarrow [\![y]\!] \\ \Leftrightarrow & \quad \{ \text{ induction hypotheses } \} \\ & \mathit{True} \end{array}$$

Conversely, in the right-to-left direction, the implication $e \Downarrow n \Rightarrow [\![e]\!] = n$ can first be rewritten in the form $\forall e \Downarrow n.\ [\![e]\!] = n$ using the shorthand notation that was introduced in Section 5, which property can then be verified by rule induction on the big-step semantics for expressions. In particular, spelling the details out we have

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

The two final conditions are then verified by simply applying the definition of $[\![\text{-}]\!]$.

**Further reading.** Big-step semantics can be useful in situations when we are only interested in the final result of execution rather than the detail of how this is performed. In this article, we primarily focus on denotational and operational approaches to semantics, but there are a variety of other approaches too, including axiomatic (Hoare, 1969), algebraic (Goguen & Malcolm, 1996), modular (Mosses, 2004), action (Mosses, 2005) and game (Abramsky & McCusker, 1999) semantics.

## 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:

$$
\begin{array}{l}
eval :: Expr \rightarrow Integer \\
eval\,(Val\,n) \quad\ = n \\
eval\,(Add\,x\,y) = eval\,x + eval\,y
\end{array}
$$

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 constructing an abstract machine for evaluating expressions.

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 this section, we show how an abstract machine for our simple expression language can be systematically derived from the evaluation function using a two-step process based on two important semantic concepts, continuations and defunctionalisation, using an approach that was pioneered by Danvy and his collaborators (Ager *et al.*, 2003*a*).

### *8.1 Step 1 – add continuations*

The first step in producing an abstract machine for the expression language is to make the order of evaluation explicit in the semantics itself. A standard technique for achieving this aim is to rewrite the semantics in *continuation-passing* style (Reynolds, 1972).

In our setting, a continuation is a function that will be applied to the result of an evaluation. For example, in the equation $eval\,(Add\;x\;y) = eval\;x + eval\;y$ from our semantics, when the first recursive call, $eval\;x$, is being evaluated, the remainder of the right-hand side of the equation, $+\;eval\;y$, can be viewed as a continuation for this evaluation, in the sense that it is the function that will be applied to the resulting value.

More formally, for our semantics $eval :: Expr \to Integer$, a continuation is a function of type $Integer \to Integer$ that will be applied to the resulting integer to give a new integer. This type can be generalised to $Integer \to a$, but we do not need the extra generality here. We capture the notion of such a continuation using the following type declaration:

**type** $Cont = Integer \to Integer$

Our aim now is to define a new semantics, $eval'$, that takes an expression and returns an integer as previously but also takes a continuation as an additional argument, which is applied to the result of evaluating the expression. That is, we seek to define a function:

$eval' :: Expr \to Cont \to Integer$

The desired behaviour of $eval'$ is captured by the following equation:

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

That is, applying $eval'$ to an expression and a continuation should give the same result as applying the continuation to the value of the expression.

At this point in most presentations, a recursive definition for $eval'$ would now be given, from which the above equation could then be proved. However, we can also view the equation as a specification for the function $eval'$, from which we then aim to discover or calculate a definition that satisfies the specification. Note that the above specification has many possible solutions, because the original semantics does not specify an evaluation order. We develop one possible solution below, but others are possible too.

To calculate the definition for $eval'$, we proceed from specification (1) by structural induction on the expression $e$. In each case, we start with the term $eval'\;e\;c$ and gradually transform it by equational reasoning, aiming to end up with a term $t$ that does not refer to the original semantics $eval$, such that we can then take $eval'\;e\;c = t$ as a defining equation for $eval'$ in this case. For the base case, $e = Val\;n$, the calculation has just two steps:

$$\begin{array}{ll}
 & eval'\;(Val\;n)\;c \\
= & \{\text{ specification (1) }\} \\
 & c\,(eval\,(Val\;n)) \\
= & \{\text{ applying } eval \text{ }\} \\
 & c\;n
\end{array}$$

Hence, we have discovered the following definition for $eval'$ in the base case:

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

That is, if the expression is an integer value, we simply apply the continuation to this value. For the inductive case, $e = Add\;x\;y$, we begin in the same way as above:

$$\begin{array}{ll}
 & eval'\;(Add\;x\;y)\;c \\
= & \{\text{ specification (1) }\}
\end{array}$$

$$\begin{array}{ll} & c\,(eval\,(Add\,x\,y)) \\ = & \quad \{\ \text{definition of } eval\ \} \\ & c\,(eval\,x + eval\,y) \end{array}$$

At this point, no further definitions can be applied. However, as we are performing an inductive calculation, we can use the induction hypotheses for the argument expressions $x$ and $y$, namely that for all $c'$ and $c''$, we have $eval'\,x\,c' = c'\,(eval\,x)$ and $eval'\,y\,c'' = c''\,(eval\,y)$. In order to use these hypotheses, we must rewrite suitable parts of the term being manipulated into the form $c'\,(eval\,x)$ and $c''\,(eval\,y)$ for some continuations $c'$ and $c''$. This can readily be achieved by abstracting over $eval\,x$ and $eval\,y$ using lambda expressions. Using these ideas, the rest of the calculation is then straightforward:

$$\begin{array}{ll} & c\,(eval\,x + eval\,y) \\ = & \quad \{\ \text{abstracting over } eval\,x\ \} \\ & (\lambda n \to c\,(n + eval\,y))\,(eval\,x) \\ = & \quad \{\ \text{induction hypothesis for } x\ \} \\ & eval'\,x\,(\lambda n \to c\,(n + eval\,y)) \\ = & \quad \{\ \text{abstracting over } eval\,y\ \} \\ & eval'\,x\,(\lambda n \to (\lambda m \to c\,(n+m))\,(eval\,y)) \\ = & \quad \{\ \text{induction hypothesis for } y\ \} \\ & eval'\,x\,(\lambda n \to eval'\,y\,(\lambda m \to c\,(n+m))) \end{array}$$

The final term now has the required form, i.e. does not refer to *eval*, and hence we have discovered the following definition for $eval'$ in the inductive case:

$$eval'\,(Add\,x\,y)\,c = eval'\,x\,(\lambda n \to eval'\,y\,(\lambda m \to c\,(n+m))$$

That is, if the expression is an addition, we evaluate the first argument $x$ and call the result $n$, then evaluate the second argument $y$ and call the result $m$, and finally apply the continuation $c$ to the sum of $n$ and $m$. In this manner, order of evaluation is now explicit in the semantics. In summary, we have calculated the following definition:

$$\begin{array}{l} eval' :: Expr \to Cont \to Integer \\ eval'\,(Val\,n)\,c \quad\;\; = c\,n \\ eval'\,(Add\,x\,y)\,c = eval'\,x\,(\lambda n \to eval'\,y\,(\lambda m \to c\,(n+m))) \end{array}$$

Finally, our original semantics can be recovered from our new semantics by substituting the identity continuation $\lambda n \to n$ into specification (1) from which $eval'$ was constructed. That is, the original semantics *eval* can now be redefined as follows:

$$\begin{array}{l} eval :: Expr \to Integer \\ eval\,e = eval'\,e\,(\lambda n \to n) \end{array}$$

### *8.2 Step 2 – defunctionalise*

We have now taken a step towards an abstract machine by making evaluation order explicit but in doing so have also taken a step away from such a machine by making the semantics into a higher-order function that takes a continuation as an additional argument. The second step is to regain the first-order nature of the original semantics by eliminating the use of continuations but retaining the explicit order of evaluation they introduced.

A standard technique for eliminating the use of functions as arguments is *defunctionalisation* (Reynolds, 1972). This technique is based upon the observation that we do not usually need the entire function-space of possible argument functions, because only a few forms of such functions are actually used in practice. Hence, we can represent the argument functions that we actually need using a datatype rather than using actual functions.

Within the definitions of the functions *eval* and *eval′*, there are only three forms of continuations that are used, namely one to end the evaluation process ($\lambda n \rightarrow n$), one to continue once the first argument of an addition has been evaluated ($\lambda n \rightarrow eval'\ y\ \cdots$) and one to add two integer results together ($\lambda m \rightarrow c\,(n+m)$). We begin by defining three combinators *halt*, *next* and *add* for constructing these forms of continuations:

$$
\begin{aligned}
&halt :: Cont\\
&halt = \lambda n \rightarrow n\\[4pt]
&next :: Expr \rightarrow Cont \rightarrow Cont\\
&next\ y\ c = \lambda n \rightarrow eval'\ y\ (add\ n\ c)\\[4pt]
&add :: Integer \rightarrow Cont \rightarrow Cont\\
&add\ n\ c = \lambda m \rightarrow c\ (n+m)
\end{aligned}
$$

In each case, free variables in the continuation become parameters of the combinator. Using the above definitions, our continuation semantics can now be rewritten as:

$$
\begin{aligned}
&eval :: Expr \rightarrow Integer\\
&eval\ e = eval'\ e\ halt\\[4pt]
&eval' :: Expr \rightarrow Cont \rightarrow Integer\\
&eval'\ (Val\ n)\quad\ \ c = c\ n\\
&eval'\ (Add\ x\ y)\ c = eval'\ x\ (next\ y\ c)
\end{aligned}
$$

The next stage in the process is to declare a first-order datatype whose constructors represent the three combinators, which can easily be achieved as follows:

$$
\begin{aligned}
&\mathbf{data}\ CONT\ \mathbf{where}\\
&\quad HALT :: CONT\\
&\quad NEXT :: Expr \rightarrow CONT \rightarrow CONT\\
&\quad ADD\ \ :: Integer \rightarrow CONT \rightarrow CONT
\end{aligned}
$$

Note that the constructors for *CONT* have the same names and types as the combinators for *Cont*, except that all the items are now capitalised. The fact that values of type *CONT* represent continuations of type *Cont* is formalised by the following translation function, which forms a denotational semantics for the new datatype:

$$
\begin{aligned}
&exec :: CONT \rightarrow Cont\\
&exec\ HALT \qquad\quad = halt\\
&exec\ (NEXT\ y\ c) = next\ y\ (exec\ c)\\
&exec\ (ADD\ n\ c)\ \ = add\ n\ (exec\ c)
\end{aligned}
$$

In the literature, this function is usually called *apply* (Reynolds, 1972), reflecting the fact that when its type is expanded to $CONT \rightarrow Integer \rightarrow Integer$, it can be viewed as applying a representation of a continuation to an integer to give another integer. The reason for using the name *exec* in our setting will become clear shortly.

Our aim now is to define a new semantics, $eval''$, that behaves in the same way as our previous semantics $eval'$, except that it uses values of type $CONT$ rather than continuations of type $Cont$. That is, we seek to define a function:

$eval'' :: Expr \to CONT \to Integer$

The desired behaviour of $eval''$ is captured by the following equation:

$$eval''\ e\ c \quad = \quad eval'\ e\ (exec\ c) \tag{2}$$

That is, applying $eval''$ to an expression and the representation of a continuation should give the same result applying $eval'$ to the expression and the continuation it represents.

As previously, to calculate the definition for $eval''$ we proceed by structural induction on the expression $e$. The base case $e = Val\ n$ is straightforward,

$$\begin{array}{ll} & eval''\ (Val\ n)\ c \\ = & \{\ \text{specification (2)}\ \} \\ & eval'\ (Val\ n)\ (exec\ c) \\ = & \{\ \text{definition of } eval'\ \} \\ & exec\ c\ n \end{array}$$

while the inductive case, $e = Add\ x\ y$, uses the definition of $exec$ to transform the term being manipulated to allow an induction hypothesis to be applied:

$$\begin{array}{ll} & eval''\ (Add\ x\ y)\ c \\ = & \{\ \text{specification (2)}\ \} \\ & eval'\ (Add\ x\ y)\ (exec\ c) \\ = & \{\ \text{definition of } eval'\ \} \\ & eval'\ x\ (next\ y\ (exec\ c)) \\ = & \{\ \text{definition of } exec\ \} \\ & eval'\ x\ (exec\ (NEXT\ y\ c)) \\ = & \{\ \text{induction hypothesis for } x\ \} \\ & eval''\ x\ (NEXT\ y\ c) \end{array}$$

However, the definition for $exec$ still refers to the previous semantics $eval'$, via its use of the combinator $next$. We can calculate a new definition for $exec$ that refers to our new semantics $eval''$ instead by simple case analysis on the $CONT$ argument (no induction required), which proceeds for the three possible forms of this argument as follows:

$$\begin{array}{ll} & exec\ HALT\ n \\ = & \{\ \text{definition of } exec\ \} \\ & halt\ n \\ = & \{\ \text{definition of } halt\ \} \\ & n \end{array}$$

and

$$\begin{array}{ll} & exec\ (NEXT\ y\ c)\ n \\ = & \{\ \text{definition of } exec\ \} \\ & next\ y\ (exec\ c)\ n \\ = & \{\ \text{definition of } next\ \} \end{array}$$

$$\begin{aligned}
&\quad \mathit{eval'}\; y\,(\mathit{add}\; n\,(\mathit{exec}\; c)) \\
=&\quad \{\ \text{definition of } \mathit{exec}\ \} \\
&\quad \mathit{eval'}\; y\,(\mathit{exec}\,(\mathit{ADD}\; n\; c)) \\
=&\quad \{\ \text{specification (2)}\ \} \\
&\quad \mathit{eval''}\; y\,(\mathit{ADD}\; n\; c)
\end{aligned}$$

and

$$\begin{aligned}
&\quad \mathit{exec}\,(\mathit{ADD}\; n\; c)\; m \\
=&\quad \{\ \text{definition of } \mathit{exec}\ \} \\
&\quad \mathit{add}\; n\,(\mathit{exec}\; c)\; m \\
=&\quad \{\ \text{definition of } \mathit{add}\ \} \\
&\quad \mathit{exec}\; c\,(n+m)
\end{aligned}$$

Finally, our original semantics *eval* for expressions can be recovered from our new semantics *eval″* by means of the following calculation:

$$\begin{aligned}
&\quad \mathit{eval}\; e \\
=&\quad \{\ \text{previous definition of } \mathit{eval}\ \} \\
&\quad \mathit{eval'}\; e\,(\lambda n \to n) \\
=&\quad \{\ \text{definition of } \mathit{halt}\ \} \\
&\quad \mathit{eval'}\; e\; \mathit{halt} \\
=&\quad \{\ \text{definition of } \mathit{exec}\ \} \\
&\quad \mathit{eval'}\; e\,(\mathit{exec}\; \mathit{HALT}) \\
=&\quad \{\ \text{specification (2)}\ \} \\
&\quad \mathit{eval''}\; e\; \mathit{HALT}
\end{aligned}$$

In summary, we have calculated the following new definitions:

$$\begin{aligned}
&\mathit{eval} :: \mathit{Expr} \to \mathit{Integer} \\
&\mathit{eval}\; e = \mathit{eval''}\; e\; \mathit{HALT}
\end{aligned}$$

$$\begin{aligned}
&\mathit{eval''} :: \mathit{Expr} \to \mathit{CONT} \to \mathit{Integer} \\
&\mathit{eval''}\,(\mathit{Val}\; n)\quad\;\; c = \mathit{exec}\; c\; n \\
&\mathit{eval''}\,(\mathit{Add}\; x\; y)\; c = \mathit{eval''}\; x\,(\mathit{NEXT}\; y\; c)
\end{aligned}$$

$$\begin{aligned}
&\mathit{exec} :: \mathit{CONT} \to \mathit{Integer} \to \mathit{Integer} \\
&\mathit{exec}\; \mathit{HALT} \qquad\;\; n\; = n \\
&\mathit{exec}\,(\mathit{NEXT}\; y\; c)\; n\; = \mathit{eval''}\; y\,(\mathit{ADD}\; n\; c) \\
&\mathit{exec}\,(\mathit{ADD}\; n\; c)\;\; m = \mathit{exec}\; c\,(n+m)
\end{aligned}$$

Together with the *CONT* type, these definitions form an abstract machine for evaluating expressions. In particular, the four components can be understood as follows:

- *CONT* is the type of *control stacks* for the machine and comprises instructions that determine how the machine should continue after evaluating the current expression. As a result, this kind of machine is sometimes called an 'eval/continue' machine. The type of control stacks could also be refactored as a list of instructions:

  $$\begin{aligned}
  &\textbf{type}\; \mathit{CONT} = [\mathit{INST}] \\
  &\textbf{data}\; \mathit{INST}\;\; = \mathit{ADD}\; \mathit{Int} \mid \mathit{NEXT}\; \mathit{Expr}
  \end{aligned}$$

However, we prefer the original definition as it arose in a systematic way and only requires the declaration of a single new type rather than two new types.

- *eval* evaluates an expression to give an integer, by simply invoking *eval''* with the given expression and the empty control stack *HALT*.
- *eval''* evaluates an expression in the context of a control stack. If the expression is an integer value, we execute the control stack using this integer as an argument. If the expression is an addition, we evaluate the first argument $x$, placing the instruction *NEXT* $y$ on top of the control stack to indicate that the second argument $y$ should be evaluated once evaluation of the first argument is completed.
- *exec* executes a control stack in the context of an integer argument. If the stack is empty, represented by the instruction *HALT*, we return the integer argument as the result of the execution. If the top of the stack is an instruction *NEXT* $y$, we evaluate the expression $y$, placing the instruction *ADD* $n$ on top of the remaining stack to indicate that the current integer argument $n$ should be added together with the result of evaluating $y$ once this is completed. Finally, if the top of the stack is an instruction *ADD* $n$, evaluation of the two arguments of an addition is complete, and we execute the remaining control stack in the context of the sum of resulting integers.

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

$$\begin{array}{cl}
 & \mathit{eval}\,(\mathit{Add}\,(\mathit{Val}\,1)\,(\mathit{Val}\,2)) \\
= & \mathit{eval}''\,(\mathit{Add}\,(\mathit{Val}\,1)\,(\mathit{Val}\,2))\,\mathit{HALT} \\
= & \mathit{eval}''\,(\mathit{Val}\,1)\,(\mathit{NEXT}\,(\mathit{Val}\,2)\,\mathit{HALT}) \\
= & \mathit{exec}\,(\mathit{NEXT}\,(\mathit{Val}\,2)\,\mathit{HALT})\,1 \\
= & \mathit{eval}''\,(\mathit{Val}\,2)\,(\mathit{ADD}\,1\,\mathit{HALT}) \\
= & \mathit{exec}\,(\mathit{ADD}\,1\,\mathit{HALT})\,2 \\
= & \mathit{exec}\,\mathit{HALT}\,3 \\
= & 3
\end{array}$$

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.

We conclude by noting that the form of control stacks used in the abstract machine is very similar to the form of contexts used in the contextual semantics in Section 6. Indeed, if we write the type of control stacks as regular algebraic datatype,

**data** *CONT* = *HALT* | *NEXT Expr CONT* | *ADD Integer CONT*

and write the type of evaluation contexts that specify the usual left-to-right evaluation order for addition from the end of Section 6 in the same style,

**data** *Con* = *Hole* | *AddL Con Expr* | *AddR Integer Con*

then we see that the two types are isomorphic, i.e. there is a one one-to-one correspondence between their values. In particular, the isomorphism is given by simply renaming the corresponding constructors and swapping the argument order in the case of *NEXT* and *AddL*. This isomorphism, which demonstrates that evaluation contexts are just defunctionalised continuations, is not specific to this particular example and illustrates a deep semantic connection that has been explored in a number of articles cited below.

**Further reading.** Reynolds' seminal paper (1972) introduced three key techniques: definitional interpreters, continuation-passing style and defunctionalisation. Danvy and his collaborators later showed how Reynolds' paper actually contained a blueprint for deriving abstract machines from evaluators (Ager *et al.*, 2003*a*) and went on to produce a series of influential papers on a range of related topics, including deriving compilers from evaluators (Ager *et al.*, 2003*b*), deriving abstract machines from small-step semantics (Danvy & Nielsen, 2004) and dualising defunctionalisation (Danvy & Millikin, 2009); additional references can be found in Danvy's invited paper (2008). 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.

This section is based upon (Hutton & Wright, 2006; Hutton & Bahr, 2016), which also show how to calculate machines for extended versions of the expression language and how the two transformation steps can be fused into a single step. Similar techniques can be used to calculate compilers for stack (Bahr & Hutton, 2015) and register machines (Hutton & Bahr, 2017; Bahr & Hutton, 2020), as well as typed (Pickard & Hutton, 2021), non-terminating (Bahr & Hutton, 2022) and concurrent (Bahr & Hutton, 2023) languages.

## 9 Summary and conclusion

In this article, we have shown 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 (Winskel, 1993; Reynolds, 1998; Pierce, 2002; Harper, 2016), summer schools including the Oregon Programming Languages Summer School (OPLSS, 2023) 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 is 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.

## Supplementary materials

For supplementary material for this article, please visit http://doi.org/10.1017/S0956796823000072

## References

Abbott, M. G., Altenkirch, T., McBride, C. & Ghani, N. (2005) $\delta$ for data: Differentiating data structures. *Fundam. Inform.* **65**(1-2), 1–28.

Abramsky, S. & Jung, A. (1994) Domain theory. In *Handbook of Logic in Computer Science*, vol. 3. Clarendon, pp. 1–168.

Abramsky, S. & McCusker, G. (1999) Game semantics. *Comput. Logic* **165**, 1–55.

Ager, M. S., Biernacki, D., Danvy, O., & Midtgaard, J. (2003a) A functional correspondence between evaluators and abstract machines. In Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming.

Ager, M. S., Biernacki, D., Danvy, O. & Midtgaard, J. (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, P. & Hutton, G. (2015) Calculating correct compilers. *J. Funct. Program.* **25**.

Bahr, P. & Hutton, G. (2020) Calculating correct compilers II: Return of the register machines. *J. Funct. Program.* **30**.

Bahr, P. & Hutton, G. (2022) Monadic compiler calculation. *Proc. ACM Program. Lang.* **6**(ICFP), 80–108.

Bahr, P. & Hutton, G. (2023) Calculating compilers for concurrency. *Proc. ACM Program. Lang.* **7**(ICFP), 740–767.

Burstall, R. (1969) Proving properties of programs by structural induction. *Comput. J.* **12**(1), 41–48.

Danvy, O. (2008) Defunctionalized interpreters for programming languages. In Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming.

Danvy, O. & Millikin, K. (2009) Refunctionalization at work. *Sci. Comput. Program.* **74**(8), 534–549.

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

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

Dybjer, P. (1994) Inductive families. *Formal Aspects Comput.* **6**(4), 440–465.

Felleisen, M. & Hieb, R. (1992) The revised report on the syntactic theories of sequential control and state. *Theoret. Comput. Sci.* **103**(2), 235–271.

Gibbons, J. & Jones, G. (1998) The under-appreciated unfold. In Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming.

Goguen, J. & Malcolm, G. (1996) *Algebraic Semantics of Imperative Programs*. MIT.

Harper, R. (2016) *Practical Foundations for Programming Languages*, 2nd ed. Cambridge University.

Hoare, T. (1969) An axiomatic basis for computer programming. *Commun. ACM* **12**, 576–583.

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

Hope, C. & Hutton, G. (2006) Accurate step counting. In *Implementation and Application of Functional Languages*. LNCS, vol. 4015. Berlin/Heidelberg: Springer, pp. 91–105.

Hu, L. & Hutton, G. (2009) Towards a verified implementation of software transactional memory. In *Trends in Functional Programming Volume 9*. Intellect, pp. 129–143.

Hu, L. & Hutton, G. (2010) Compiling concurrency correctly: Cutting out the middle man. In *Trends in Functional Programming Volume 10*. Intellect, pp. 17–32.

Huet, G. (1997) The zipper. *J. Funct. Program.* **7**(5), 549–554.

Hutton, G. (1998) Fold and unfold for program semantics. In Proceedings of the 3rd International Conference on Functional Programming.

Hutton, G. & Bahr, P. (2016) Cutting out continuations. In *A List of Successes That Can Change the World*. LNCS, vol. 9600. Springer, pp. 187–200.

Hutton, G. & Bahr, P. (2017) Compiling a 50-year journey. *J. Funct. Program.* **27**.

Hutton, G. & Wright, J. (2004) Compiling exceptions correctly. In Proceedings of the 7th International Conference on Mathematics of Program Construction. LNCS, vol. 3125. Springer.

Hutton, G. & Wright, J. (2006) Calculating an Exceptional Machine. In *Trends in Functional Programming Volume 5*. Intellect, pp. 49–64.

Hutton, G. & Wright, J. (2007) What is the meaning of these constant interruptions? *J. Funct. Program.* **17**(6), 777–792.

Kahn, G. (1987) Natural semantics. In Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science.

Landin, P. (1964) The mechanical evaluation of expressions. *Comput. J.* **6**, 308–320.

McBride, C. (2008) Clowns to the left of me, jokers to the right: Dissecting data structures. In Proceedings of the Symposium on Principles of Programming Languages.

McCarthy, J. & Painter, J. (1967) Correctness of a compiler for arithmetic expressions. In *Mathematical Aspects of Computer Science*. Proceedings of Symposia in Applied Mathematics, vol. 19. American Mathematical Society, pp. 33–41.

Meijer, E., Fokkinga, M. & Paterson, R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. In Proceedings of the Conference on Functional Programming and Computer Architecture.

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

Milner, R. (1999) *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University.

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

Mosses, P. (2004) Modular structural operational semantics. *J. Logic Algebraic Program.* **60-61**, 195–228.

Mosses, P. (2005) *Action Semantics*. Cambridge University.

Norell, U. (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. (2023) *Oregon Programming Languages Summer School*. Available at: `https://www.cs.uoregon.edu/research/summerschool/archives.html`.

Pickard, M. & Hutton, G. (2021) Calculating dependently-typed compilers. *Proc. ACM Program. Lang.* **5**(ICFP), 1–27.

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

Plotkin, G. (1981) *A Structured Approach to Operational Semantics*. Report DAIMI-FN-19. Computer Science Department, Aarhus University, Denmark, pp. 3–15.

Plotkin, G. (2004) The origins of structural operational semantics. *J. Logic Algebraic Program.* **60-61**.

Reynolds, J. C. (1972) Definitional interpreters for higher-order programming languages. In Proceedings of the ACM Annual Conference.

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

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

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

Wadler, P. (1998) *The Expression Problem*. Available at: http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.

Wand, M. (1982) Deriving target code as a representation of continuation semantics. *ACM Trans. Program. Lang. Syst.* **4**(3), 496–517.

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

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