Under consideration for publication in the Journal of Functional Programming

FUNCTIONAL PEARL
It's Easy As 1,2,3

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

Abstract

For more than twenty years now we have been using the language of integers and addition as a minimal setting in which to explore different aspects of programming language semantics. However, this language has never been the subject of an article in its own right. This article fills this gap, by showing how a range of semantic ideas can be presented in a simple manner through the lens of integers and addition. 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, before moving on to a more general setting. This article is about such an example that has become a central focus of our research on programming languages and their semantics during the last twenty years: the language of simple arithmetic expressions built up from integer values using an addition operator.

In the beginning, we used this language as a means to *explain* semantic ideas in a simple manner, but over time it also developed into a mechanism to help *discover* new ideas. To date it has been used in fifteen of our research articles, by five of our students in their doctoral dissertations, and in two textbooks. The purpose of this article is to consolidate the approach so that others may benefit from it too in their own work, whether they be students learning about semantics, or researchers advancing the state-of-the-art.

Using a minimal language to explore semantic ideas is an example of applying Occam's Razor (Duignan, 2018), a philosophical principle that gives precedence to simplicity by removing unnecessary details. As a result, the language of integers and addition is sometimes called Hutton's Razor. While this language does not provide features that are necessary for actual programming, it *does* provide just enough structure to explore many concepts from semantics. In particular, the integers provide a simple notion of 'value', and the addition operator provides a simple notion of 'computation'. More technically, the language is a degenerate example of a monad (integers and addition are the free group over one generator, and free constructions are monadic), an approach to modelling effects that is widely used in programming (Wadler, 1992) and semantics (Moggi, 1991).

Of course, one could consider an even simpler language, such as (Peano) natural numbers built up from the value zero using a successor operator. However, this language may be *too* simple for some semantic applications, because natural numbers have a list-like structure (one-way branching), whereas expressions formed from integers and addition are tree-like (two-way branching). Similarly, one might ask whether subtraction is a better choice of basic operation than addition? However, we have found that in a number of applications associativity is an important property, and while addition is associative, subtraction is not. More generally, using subtraction rather than addition would break the connection with monads, which requires that the basic computation mechanism is associative.

One might also consider some notion of variable binding, as used for example in the lambda calculus to define functions. Binding is an important topic, but its addition to a language usually requires that we then consider a range of others concepts such as environments, fresh names, alpha equivalence, and variable renaming. These are all important, but our experience time and time again is that there is much to be gained by first focusing on the simpler language that just comprises integers and addition. Once the basic ideas are developed in this setting, one can then extend the language with other features of interest, a stepwise approach that has proved useful in many aspects of our work.

Starting with a simple language first can allow us to focus on the essence of a problem, and see things clearly that may be obscured in more complicated settings. For example, different approaches to specifying the semantics of languages can readily be explained in an elementary manner in this setting. Moreover, the simplicity of the language has also been instrumental to a number of research advances that may have been difficult to discover in more sophisticated settings. For example, it was key to our discovery of new approaches to calculating implementations of semantics using elementary techniques. We will see examples of both the explanatory and discovery aspects in this article.

The article is written in a tutorial style that does not assume prior knowledge of semantics, but we hope that more experienced readers will also find new ideas and inspiration for their own work. Note that we are not aiming to provide a comprehensive account of semantics in either breadth or depth, or references to the literature, but rather to summarise the basic ideas and benefits of our approach, and provide pointers to further reading. We use Haskell throughout as a meta-language to implement semantic ideas, which helps to make the ideas more concrete, and allows them to be executed and tested.

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 used to disambiguate expressions if required. The grammar for expressions can also be directly translated into a Haskell datatype declaration:

$$\textbf{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}$$

For example, the expression $1+2$ can be represented by the term $\mathit{Add}\ (\mathit{Val}\ 1)\ (\mathit{Val}\ 2)$. 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 (Schmidt, 1986), 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* $[\![\ ]\!] : T \to V$ that maps terms to their meaning as values. Moreover, the valuation function must be *compositional*, in the sense that the meaning of a compound term is defined purely in terms of the meaning of its subterms, which property 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 *Int* of integers, and $[\![\ ]\!] : \mathit{Expr} \to \mathit{Int}$ as the evaluation function for expressions defined recursively as follows:

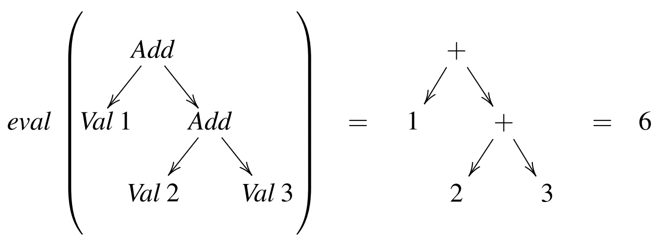
$$
\begin{aligned}
[\![\mathit{Val}\ n]\!] &= n \\
[\![\mathit{Add}\ x\ y]\!] &= [\![x]\!] + [\![y]\!]
\end{aligned}
$$

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:

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

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



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 non-recursively as simply replacing the constructors for expressions by other functions. In this manner, a denotational semantics can be viewed as an evaluation function that is defined by *folding* over the syntax of the source language.

Note that if the grammar $E ::= \mathbb{Z} \mid E+E$ was used as our source language, as opposed to the type *Expr*, then the 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.

Finally, we remark 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. Making this explicit requires the introduction of additional structure into the evaluation function, which we will consider later on in the article.

Further reading Generalising fold operators (Meijer & Hutton, 1995); expressing denotational semantics using folds (Hutton, 1998); extending the language with features such as exceptions (Hutton & Wright, 2004), interrupts (Hutton & Wright, 2007), transactions (Hu & Hutton, 2009) and lambda calculus (Bahr & Hutton, 2015); defining modular denotational semantics (Day & Hutton, 2012; Day & Hutton, 2013).

4 Operational 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 semantic setting. There are two basic forms of operational semantics: *small-step*, which describes the individual steps of execution, and *big-step* semantics, which describes the overall results of execution.

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

Arithmetic expressions also have a simple small-step operational semantics, given by taking S as the Haskell type *Expr* of expressions, and $\rightarrow \subseteq \mathit{Expr} \times \mathit{Expr}$ as the transition relation defined by the following three inference rules:

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

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

This first rule states that two values can be added together to give a single value, and the last two rules permit the first to be applied to either argument of an addition. For example, the expression $(1+2)+(3+4)$, written in normal syntax for brevity, has two possible transitions, because the first rule can be applied to either argument of the addition:

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

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:

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

The transition relation can also be translated directly into a Haskell function definition, by using the comprehension notation to return the list of all expressions that can be reached from a given expression by performing a single execution step:

$$\begin{array}{ll} \mathit{trans} :: \mathit{Expr} \rightarrow [\mathit{Expr}] & \\ \mathit{trans}\ (\mathit{Val}\ n) & = [\,] \\ \mathit{trans}\ (\mathit{Add}\ (\mathit{Val}\ n)\ (\mathit{Val}\ m)) & = [\mathit{Val}\ (n+m)] \\ \mathit{trans}\ (\mathit{Add}\ x\ y) & = [\mathit{Add}\ x'\ y \mid x' \leftarrow \mathit{trans}\ x] +\!\!+ [\mathit{Add}\ x\ y' \mid y' \leftarrow \mathit{trans}\ y] \end{array}$$

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:

$$\mathbf{data}\ \mathit{Tree}\ a = \mathit{Node}\ a\ [\mathit{Tree}\ a]$$

$$\begin{array}{l} \mathit{exec} :: \mathit{Expr} \rightarrow \mathit{Tree}\ \mathit{Expr} \\ \mathit{exec}\ e = \mathit{Node}\ e\ [\mathit{exec}\ e' \mid e' \leftarrow \mathit{trans}\ e] \end{array}$$

From this definition, we see that an expression is executed by taking the expression itself as the root of the tree, or equivalently, by applying the identity function *id* to the expression, 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 non-recursively 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.

Note that if the grammar E was used as our source language rather than the type *Expr*, then the first inference rule for the semantics would have the form

$$\frac{}{n+m \;\rightarrow\; n+m}$$

which 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.

Finally, we remark that the above semantics for expressions does not specify the order of evaluation. However, if we wish to do so, 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:

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

Further reading Generalising unfold operators (Meijer & Hutton, 1995); expressing operational semantics using unfolds (Hutton, 1998); extending the language with features such as transactions (Hu & Hutton, 2009) and non-determinism (Hu & Hutton, 2010); defining modular operational semantics (Jaskelioff *et al.*, 2008).

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 so-called *structural* rules that express how the basic rules can be applied in larger terms. Separating these two forms of rules gives rise to a *contextual* semantics (Felleisen & Hieb, 1992).

Informally, a *context* in this setting is a term with a 'hole', usually written as $[-]$, which can be 'filled in' 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:

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

In this example, 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 $1+2 \rightarrow 3$ 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:

$\textbf{data}\ \mathit{Con} = \mathit{Hole} \mid \mathit{AddL}\ \mathit{Con}\ \mathit{Expr} \mid \mathit{AddR}\ \mathit{Expr}\ \mathit{Con}$

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

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

That is, if the context is a hole we simply substitute the hole by the given expression, otherwise we recurse on the left or right side of an addition as appropriate. Using this notation, we can now redefine the small-step semantics for arithmetic expressions in contextual style, by means of the following two inference rules:

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

The first rule captures the basic behaviour of addition, as previously. In turn, second rule allows the first 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 wish to extend the language with extra features, this usually only requires adding new basic rules and extending the type for contexts, but typically does not require adding new structural rules.

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

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

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

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

More formally, the behaviour of this definition can be captured as follows: a pair (c, x) comprising a context c and an expression x is an element of the list *decompose e* precisely when *subst c x* $= e$. Using these two functions, the contextual transition relation $\rightarrow$ can also then be translated into a Haskell function definition:

```
trans :: Expr → [Expr]
trans (Add (Val n) (Val m)) = [Val (n + m)]
trans e                     = [subst c x' | (c, x) ← tail (decompose e), x' ← trans x]
```

The first equation implements the basic rule for addition, while the second implements the contextual rule by first decomposing the expression into all possible pairs of contexts and expressions, then recursively considering the transitions that can be made by each component expression, and finally, substituting the resulting expressions back into the context. Note that to avoid the *trans* function simply looping on the input expression e, the first decomposition (Hole, e) is discarded by taking the *tail* of the list.

Further reading Contexts are related to a number of other important ideas in programming and semantics, including the use of zippers to navigate around recursive data structures (Huet, 1997), the notion of differentiation for datatypes (Abbott *et al.*, 2005), the use of continuations to express control flow (Reynolds, 1972), and implementing programming languages using abstract machines (Ager *et al.*, 2003).

6 Big-Step Semantics

Whereas small-step semantics focus on single steps of execution, *big-step* semantics specify how terms can be executed to completion 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* $\Downarrow \subseteq T \times V$ that relates each term to all values that can be reached by fully executing the term. If $(t, v) \in \Downarrow$ 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 the set V as the Haskell type *Int* of integers, and $\Downarrow \subseteq Expr \times Int$ as the evaluation relation defined 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 term to completion:

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

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. Note, however, that 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, such as the lambda calculus (Bahr & Hutton, 2015). Moreover, whereas a denotational semantics can only return a single value, a big-step semantics can return multiple values if desired. This additional flexibility can sometimes be useful, such as when considering non-deterministic languages (Hutton & Wright, 2007).

Further reading Big-step operational semantics tends to be more widely used than small-step, because in many applications we are only interested in the result of execution. However, the small-step approach can be useful when the fine structure of execution is important, such as when considering efficiency (Hope & Hutton, 2006), abstract machines (Hutton & Wright, 2006) or concurrent languages (Hu & Hutton, 2009).

7 Rule Induction

Once a semantics for a language has been defined, it can then be used as the basis for proving properties of the language. Given that terms and their semantics are built up inductively, such proofs usually proceed using some form of induction. In the case of denotational semantics, the basic proof technique is structural induction, which most computer scientists are familiar with. For operational semantics, the basic technique is *rule induction*, but unfortunately many textbooks and courses on semantics gloss over the details. In this section we present the details for our simple expression language.

We introduce the idea of rule induction by first considering the special case when a set X is inductively defined by two rules of the following form:

$$\frac{}{a \in X} \qquad\qquad \frac{x \in X}{f(x) \in X}$$

The first rule, the base case for the definition, states that some given value a is in the set X. The second rule, the inductive case, states that for any value x in X, then we have $f(x)$ in X for some given function f. Note that unlike the special case when the set X is a recursively defined datatype, which is known as a *free* datatype, there is no requirement that the function f must be injective, or that the value a is not in the range of f. This is a key difference between rule induction and structural induction.

Given such an inductively defined set X, the principle of rule induction states that in order to prove that some property P holds for all elements of X, it suffices to show that P holds for the value a in X, the *base* case, and that if P holds for any element x in X then it also holds for $f(x)$, the *inductive* case. That is, we have the following proof rule:

$$\frac{P(a) \qquad \forall x \in X.\, P(x) \;\Rightarrow\; P(f(x))}{\forall x \in X.\, P(x)}$$

This basic scheme can easily be generalised multiple base and inductive cases, to rules to 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:

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

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

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

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

That is, we must show that the property P holds for the transition defined by base rule, 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.

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:

$$\forall x \to x'.\ [\![x]\!] = [\![x']\!]$$

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:

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

Each condition can then be verified by a simple calculation:

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

and

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

and

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

We conclude with three 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, it is based on the structure of the definition of the transition relation, which is the key structure here, rather than the syntactic structure of expressions, which is secondary.

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 $x \;\rightarrow\; x'$, and then immediately stating and verifying the three sufficient conditions.

And finally, because all of our semantics are also implemented in Haskell, we can use QuickCheck (Claessen & Hughes, 2000) to perform random testing of properties. For example, the property that making a transition does not change the denotation of an expression can be expressed in Haskell using a list comprehension:

$$\begin{aligned}&\mathit{prop} :: \mathit{Expr} \rightarrow \mathit{Bool}\\&\mathit{prop}\;x = \mathit{and}\;[\mathit{eval}\;x == \mathit{eval}\;x' \mid x' \leftarrow \mathit{trans}\;x]\end{aligned}$$

If we now use the Quickcheck library to write a random generator for expressions, it can then be used to automatically test the property on a range of examples:

$$\begin{aligned}&> \mathit{quickCheck}\;\mathit{prop}\\&{+}{+}{+}\;\mathit{OK},\;\mathit{passed}\;100\;\mathit{tests}.\end{aligned}$$

The idea of testing semantic properties in this lightweight manner has proved invaluable in our research work. For example, in our work on compiler correctness, we routinely use QuickCheck to debug our definitions and theorems prior to formal proofs, and this has often revealed subtle errors and omissions in our definitions.

Further reading Using rule induction to verify the equivalence of small and big-step operational semantics for versions of the lambda calculus that count evaluation steps (Hope, 2008) or support a form of non-deterministic choice (Moran, 1998); validating compiler correctness results using QuickCheck (Hu & Hutton, 2009).

8 Abstract Machines

All of the examples that 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* (Diehl *et al.*, 2000) for evaluating expressions in a manner that precisely defines the order of evaluation.

We begin by recalling the following simple evaluation function:

$$\begin{aligned}&\mathit{eval} :: \mathit{Expr} \rightarrow \mathit{Int}\\&\mathit{eval}\;(\mathit{Val}\;n) \quad\;\; = n\\&\mathit{eval}\;(\mathit{Add}\;x\;y) = \mathit{eval}\;x + \mathit{eval}\;y\end{aligned}$$

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 language in which the evaluator is written, in this case Haskell. If desired, the order of evaluation can be made explicit by defining an 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 for evaluating expressions of type *Expr* can be given by three components: a type *Cont* of control stacks, a function $eval' :: Expr \to Cont \to Int$ that evaluates an expression and then continues by executing the given control stack, and finally, a function $exec :: Cont \to Int \to Int$ 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.

To calculate the 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:

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

The resulting term $exec\ c\ n$ already has the required form (does not refer to *eval*), from which 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 by applying the evaluation function:

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

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,

$$ADD :: Int \to Cont \to Cont$$

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

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

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 first is completed. In this manner, the definition makes explicit that addition is evaluated in left-to-right order.

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

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

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

$$
\begin{array}{l}
\textbf{data}\ Cont = HALT \mid EVAL\ Expr\ Cont \mid ADD\ Int\ Cont \\[6pt]
eval :: Expr \to Int \\
eval\ e = eval'\ e\ HALT \\[6pt]
eval' :: Expr \to Cont \to Int \\
eval'\ (Val\ n)\qquad c\ = exec\ c\ n \\
eval'\ (Add\ x\ y)\quad c\ = eval'\ x\ (EVAL\ y\ c) \\[6pt]
exec :: Cont \to Int \to Int \\
exec\ HALT\qquad\quad n\ = n \\
exec\ (EVAL\ y\ c)\ n\ = eval'\ y\ (ADD\ n\ c) \\
exec\ (ADD\ n\ c)\ \ m = exec\ c\ (n+m)
\end{array}
$$

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 $(2+3)+4$ we have:

$$
\begin{array}{ll}
 & eval\ (Add\ (Add\ (Val\ 2)\ (Val\ 3))\ (Val\ 4)) \\
= & eval'\ (Add\ (Add\ (Val\ 2)\ (Val\ 3))\ (Val\ 4))\ HALT \\
= & eval'\ (Add\ (Val\ 2)\ (Val\ 3))\ (EVAL\ (Val\ 4)\ HALT) \\
= & eval'\ (Val\ 2)\ (EVAL\ (Val\ 3)\ (EVAL\ (Val\ 4)\ HALT)) \\
= & exec\ (EVAL\ (Val\ 3)\ (EVAL\ (Val\ 4)\ HALT))\ 2 \\
= & eval'\ (Val\ 3)\ (ADD\ 2\ (EVAL\ (Val\ 4)\ HALT)) \\
= & exec\ (ADD\ 2\ (EVAL\ (Val\ 4)\ HALT))\ 3 \\
= & exec\ (EVAL\ (Val\ 4)\ HALT)\ 5 \\
= & eval'\ (Val\ 4)\ (ADD\ 5\ HALT) \\
= & exec\ (ADD\ 5\ HALT)\ 4 \\
= & exec\ HALT\ 9 \\
= & 9
\end{array}
$$

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. 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. Focusing on the simple language of integers and addition was key to our discovery of this simpler approach. In this setting, it's easy as 1,2,3.

Further reading Calculating abstract machines (Hutton & Wright, 2006; Hutton & Bahr, 2016); verifying compiler correctness for languages with exceptions (Hutton & Wright, 2004), interrupts (Hutton & Wright, 2007), transactions (Hu & Hutton, 2009) and concurrency (Hu & Hutton, 2010); calculating compilers for stack (Bahr & Hutton, 2015; Hutton, 2016) and register machines (Hutton & Bahr, 2017; Bahr & Hutton, 2020).

Acknowledgements

The author was funded by EPSRC grant EP/P00587X/1, *Unified Reasoning About Program Correctness and Efficiency*, for which funding is gratefully acknowledged.

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).

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

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**.

Claessen, Koen, & Hughes, John. (2000). QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *International Conference on Functional Programming*.

Day, Laurence, & Hutton, Graham. (2012). Towards Modular Compilers For Effects. *Trends in Functional Programming Volume 12*. LNCS, vol. 7193. Springer.

Day, Laurence E., & Hutton, Graham. (2013). Compilation à la Carte. *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages*.

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`.

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

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. (2016). *Programming in Haskell*. Cambridge University Press.

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).

Jaskelioff, Mauro, Ghani, Neil, & Hutton, Graham. (2008). Modularity and Implementation of Mathematical Operational Semantics. *Proceedings of the Workshop on Mathematically Structured Functional Programming*.

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

Meijer, Erik, & Hutton, Graham. (1995). Bananas in Space: Extending Fold and Unfold to Exponential Types. *Proceedings of the 7th International Conference on Functional Programming and Computer Architecture*.

Moggi, Eugenio. (1991). Notions of Computation and Monads. *Information and Computation*, **93**(1).

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

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

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

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

Wadler, Philip. (1992). The Essence of Functional Programming. *Proceedings of the Symposium on Principles of Programming Languages*.