Programming Language Semantics

It’s Easy As 1,2,3

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

Abstract

For more than twenty years now I’ve been using the language of integers and addition as a minimal setting in which to explore different aspects of programming language semantics. This article brings this work together by showing how a range of semantic concepts 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 my work 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, I 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 seventeen of my research articles (Meijer & Hutton, 1995; Hutton, 1998; Hutton & Wright, 2004; Hutton & Wright, 2006; Hope & Hutton, 2006; Hutton & Wright, 2007; Fasel et al., 2008; Hu & Hutton, 2009; Hu & Hutton, 2010; Day & Hutton, 2012; Bahr & Hutton, 2013; Day & Hutton, 2013; Bahr & Hutton, 2015; Bahr & Hutton, 2016; Bahr & Hutton, 2017; Bahr & Hutton, 2020; Pickard & Hutton, 2021; Bahr & Hutton, 2021) and in two textbooks (Hutton, 2007; Hutton, 2016). The purpose of this article is to consolidate this work so that others may benefit from it too, 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 Occam’s Razor (Duignan, 2018), a philosophical principle that favours the simplest explanation for a phenomenon. While the language of integers and addition does not provide features that are necessary for actual programming, it does provide just enough structure to explain many concepts from semantics. In particular, the integers provide a simple notion of ‘value’, and the addition operator provides a simple notion of ‘computation’. This language has been used by many authors in the past, such as McCarthy and Painter (1967), Wand (1982) and Wadler (1998), to name but a few. However, the programme of work on which this article is based is the first time it has been used as a dedicated semantic tool.
Of course, one could consider an even simpler language, such as natural numbers built up from 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). One might also consider some notion of variable binding, as used for example in the lambda calculus. 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 my 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 my work.

Starting with a simple language first allows us to focus on the essence of a problem, and see things clearly that may be obscured in more complicated settings. For example, different semantic approaches 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 the development of new techniques for calculating correct compilers [Bahr & Hutton, 2015, Bahr & Hutton, 2020].

The article is written in a tutorial style that does not assume prior knowledge of semantics, but I hope that experienced readers will also find useful ideas and inspiration for their own work. Beginners may wish to initially focus on sections 1–7, which provide an overview of a range of semantic concepts, while those with more experience may wish to focus on sections 8 and 9, which present two extended examples. Note that the article is not aiming to provide a comprehensive account of semantics in either breadth or depth, but rather to summarise the basic ideas and benefits of our 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 and tested. All of the Haskell code is available as online 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 \texttt{Integer} of arbitrary precision integers:

\[
\text{data} \ Expr = \text{Val} \ Integer | \text{Add} \ Expr \ Expr
\]
For example, the expression $1 + 2$ can be 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 (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 of type $T \rightarrow V$ that maps terms to their meaning as values. The valuation function is typically written by enclosing a term in semantic brackets, writing $Jt$ for result of applying the valuation function to the term $t$. The semantic brackets are also known as Oxford or Strachey brackets, after the pioneering work of Christopher Strachey (Scott & Strachey, 1971).

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 is important because it supports the use of structural induction to reason about the semantics. When the set of semantic values is clear, a denotational semantics is often identified with the underlying valuation function.

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

$$
\begin{align*}
[JVal\ n] & = n \\
[JAdd\ x\ y] & = [x] + [y]
\end{align*}
$$

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:

```haskell
eval :: Expr -> Integer
eval (Val n) = n
eval (Add x y) = eval x + eval y
```

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...
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:

\[
\text{eval} :: \text{Expr} \rightarrow \text{Integer} \\
\text{eval} = \text{fold id (+)}
\]

The fold operator captures the idea of replacing the constructors of the language by other functions, in our case replacing `Val` and `Add` by some functions `f` and `g`:

\[
\text{fold} :: (\text{Integer} \rightarrow a) \rightarrow (a \rightarrow a \rightarrow a) \rightarrow \text{Expr} \rightarrow a \\
\text{fold f g (Val n)} = f n \\
\text{fold f g (Add x y)} = g (\text{fold f g x}) (\text{fold f g y})
\]

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

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

as our source language, rather than the type `Expr`, then the denotational semantics would have the following form:

\[
\begin{align*}
[n] &= n \\
[x + y] &= [x] + [y]
\end{align*}
\]

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

And secondly, note that the above semantics for arithmetic expressions does not specify the order of evaluation, that is, the order in which the two arguments of addition should be evaluated. Making this explicit requires the introduction of additional structure into the semantics, which we will discuss later on when we consider abstract machines.

**Further reading** Winskel’s (1993) textbook on semantics contains an accessible introduction to the denotational approach. The idea of defining denotational semantics using fold
operators is explored further in (Hutton, 1998). The simple expression language has also been used as a basis for studying a range of other features, including exceptions (Hutton & Wright, 2004), interrupts (Hutton & Wright, 2007), transactions (Hu & Hutton, 2009), non-determinism (Hu & Hutton, 2010) and state (Bahr & Hutton, 2015).

4 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 machine model. 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, and will return to the big-step approach later on.

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 $s \rightarrow s'$. More general notions of transition relation are sometimes used, but this simple notion suffices for our purposes here. When the set of states is clear, an operational semantics is often identified with the underlying transition relation.

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

$$\begin{align*}
&\text{Add } (\text{Val } n) (\text{Val } m) \rightarrow \text{Val } (n + m) \\
&x \rightarrow x' \\
&\text{Add } x \ y \rightarrow \text{Add } x' \ y \\
&y \rightarrow y' \\
&\text{Add } x \ y \rightarrow \text{Add } x \ y'
\end{align*}$$

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

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

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
Graham Hutton

gives rise to the following tree, which captures the two possible execution paths:

```
(1 + 2) + (3 + 4)  
  ↓         ↓         
3 + (3 + 4)        (1 + 2) + 7
    ↓   ↓
3 + 7 3 + 7
     ↓   ↓
10 10
```

The transition relation can also be translated into a Haskell function definition, by exploiting the fact that a relation can be represented as a non-deterministic function that returns all possible values that are related to a given value. Using the 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 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:

```
unfold :: (a → b) → (a → [a]) → a → Tree b
unfold f g x = Node (f x) [unfold f g x' | x' ← g x]
```

In summary, whereas denotational semantics corresponds to ‘folding over syntax trees’, operational semantics corresponds to ‘unfolding to transition trees’. Thinking about semantics in terms of recursion operators reveals a duality that might otherwise have been missed, and still isn’t as widely known as it should be.
It’s Easy As 1,2,3

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:

\[ n + m \rightarrow n + m \]

However, this rule would be rather confusing unless we introduced some additional nota-
tion 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:

\[ y \rightarrow y' \]

\[ \text{Add} \ (\text{Val} \ n) \ y \rightarrow \text{Add} \ (\text{Val} \ n) \ y' \]

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

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

Further reading The idea of expressing operational semantics using unfold operators is
explored in further detail in (Hutton, 1998). In this article we primarily focus on denota-
tional and operational approaches to semantics, but there are a variety of other approaches,
including axiomatic (Hoare, 1969), algebraic (Goguen & Malcolm, 1996), action (Mosses,
2005) and game (Abramsky & McCusker, 1999) semantics.

5 Contextual Semantics

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

Informally, a context in this setting is a term with a ‘hole’, usually written as [−], which
can be ‘filled 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:

\[(1 + 2) + (3 + 4) \rightarrow 3 + (3 + 4)\]

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 \(1 + 2 \rightarrow 3\) 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:

\[C ::= [-] | C + E | E + C\]

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:

```haskell
data Con = Hole | AddL Con Expr | AddR Expr 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 \(c[x]\):

\[
\begin{align*}
\text{Hole} & \quad [x] = x \\
(\text{AddL} \ c \ e)[x] & \quad = \text{Add} (c[x]) \ e \\
(\text{AddR} \ e \ c)[x] & \quad = \text{Add} \ e (c[x])
\end{align*}
\]

That is, if the context is a hole we simply substitute by 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 substitution, which uses Haskell syntax for contexts and expressions. As usual, we’ll see shortly how it can be implemented in Haskell itself.

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

\[
\begin{align*}
\text{Add} (\text{Val} \ n) (\text{Val} \ m) & \quad \rightarrow \text{Val} (n + m) \\
\text{x} & \quad \rightarrow \text{x}' \\
\text{c[x]} & \quad \rightarrow \text{c[x']}
\end{align*}
\]

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

The contextual semantics can readily be translated into Haskell. Defining 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 splits an expression into all possible pairs of contexts and expressions, can be defined using the comprehension notation:
It’s Easy As 1,2,3

\[\text{split} :: \text{Expr} \rightarrow \{(\text{Con}, \text{Expr})\}\]
\[\text{split } e = (\text{Hole}, e) : \text{case } e \text{ of}\]
\[\text{Val } n \rightarrow []\]
\[\text{Add } x \text{ y} \rightarrow [(\text{AddL } c \text{ y}, e) \mid (c, e) \leftarrow \text{split } x]++]\]
\[[(\text{AddR } x \text{ c}, e) \mid (c, e) \leftarrow \text{split } y]\]

The behaviour of this definition can be formally characterised as follows: a pair \((c, x)\) comprising a context \(c\) and an expression \(x\) is an element of the list \(\text{split } e\) precisely when \(\text{subst } c \ x = e\). Using these two functions, the contextual transition relation can also then be translated into a Haskell function definition, which returns the list of all expressions that can be reached by performing a single execution step:

\[\text{trans} :: \text{Expr} \rightarrow \text{[Expr]}\]
\[\text{trans } (\text{Add } (\text{Val } n) \ (\text{Val } m)) = \text{[Val } (n + m)]\]
\[\text{trans } e = \text{[subst } c \ x' \mid (c, x) \leftarrow \text{tail } (\text{split } e), x' \leftarrow \text{trans } x]\]

The first equation implements the basic rule for addition, while the second implements the contextual rule by first splitting the given expression into all possible context and expression pairs, 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 \(\text{trans}\) function simply looping on the input expression \(e\), the first decomposition \((\text{Hole}, e)\) is discarded by taking the tail of the list.

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

6 Big-Step Semantics

Whereas small-step semantics focus on single execution steps, big-step semantics specify how terms can be fully executed in one large step. Formally, a big-step operational semantics, also known as a natural semantics (Kahn, 1987), for a language \(T\) of syntactic terms comprises two components: a set \(V\) of values, and an evaluation relation between \(T\) and \(V\) that relates each term to all values that can be reached by fully executing the term. If a term \(t\) and a value \(v\) are related, we say that \(t\) can evaluate to \(v\), and write this as \(t \downarrow v\).

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

- \(\text{Val } n \downarrow n\)
- \(\text{Add } x \text{ y} \downarrow n + m\)

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:

\[
\text{eval} :: \text{Expr} \to [\text{Integer}]
\]

\[
\text{eval} (\text{Val} \ n) = [n]
\]

\[
\text{eval} (\text{Add} \ x \ y) = [n + m \mid n \leftarrow \text{eval} \ x, m \leftarrow \text{eval} \ y]
\]

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

## 7 Rule Induction

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

We introduce the idea of rule induction using a simple example. Suppose that a set \( S \) of non-empty strings of stars is inductively defined by the following two rules:

\[
\begin{align*}
\star & \in S \\
ss & \in S
\end{align*}
\]

The first rule, the base case, states that a single star \( \star \) is in the set \( S \). The second rule, the inductive case, states that for any string \( s \) in \( S \), the string \( ss \) formed by concatenating \( s \) with itself is also in \( S \). Moreover, the inductive nature of the definition means that there is nothing in \( S \) beyond those strings obtained by applying these rules a finite number of times, which is sometimes called the ‘extremal clause’ of the definition.
It’s Easy As 1,2,3

Note that unlike sets that correspond to recursively defined datatypes, such as natural numbers built up from zero using a successor operation, there is not a unique way to decompose each element of \( S \) in terms of \( \star \) and concatenation. For example, the string \( \star \star \star \) could be decomposed as \( (\star \star) \star \) or \( \star (\star \star) \). In general, all that we know is that \( \star \) and concatenation are jointly surjective, i.e. all elements of \( S \) can be generated using them. This is a key difference between rule induction and structural induction.

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

\[
\begin{array}{c}
P(\star) \\
\forall s \in S. P(s) \Rightarrow P(ss)
\end{array}
\]

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

\[
\begin{align*}
\text{Add} (\text{Val} n) (\text{Val} m) & \rightarrow \text{Val} (n + m) \\
x \rightarrow x' & \text{Add } x y \rightarrow x' y \\
y \rightarrow y' & \text{Add } x y \rightarrow x' y'
\end{align*}
\]

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

\[
\begin{align*}
\forall x \rightarrow x'. P(x, x') & \Rightarrow P(\text{Add } x y, \text{Add } x y') \\
\forall y \rightarrow y'. P(y, y') & \Rightarrow P(\text{Add } x y, \text{Add } x y') \\
\forall x \rightarrow x'. P(x, x')
\end{align*}
\]

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 \( \forall x \rightarrow y. P(x, y) \) as shorthand for \( \forall x, y. x \rightarrow y \Rightarrow P(x, y) \).

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

\[
\forall x \rightarrow x'. [x] = [x']
\]

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:

\[
\begin{align*}
\forall x \rightarrow x'. [x] &= [x'] \\
\Leftrightarrow & \{ \text{define } P(x, x') \leftrightarrow [x] = [x'] \} \\
\forall x \rightarrow x'. P(x, x') & \Leftrightarrow \{ \text{rule induction for } \rightarrow \}\}
\]
The three final conditions can then be verified by simple calculations over the denotational semantics for expressions, which we include below for completeness:

\[
\begin{align*}
\llbracket \text{Add} (\text{Val} \ n) (\text{Val} \ m) \rrbracket &= \{ \text{definition of } \llbracket \ ] \rrbracket \} \\
\llbracket \text{Val} \ n \rrbracket + \llbracket \text{Val} \ m \rrbracket &= \{ \text{definition of } \llbracket \ ] \rrbracket \} \\
n + m &= \{ \text{definition of } \llbracket \ ] \rrbracket \} \\
\llbracket \text{Val} \ (n + m) \rrbracket &
\end{align*}
\]

and

\[
\begin{align*}
\llbracket \text{Add} \ x \ y \rrbracket &= \{ \text{definition of } \llbracket \ ] \rrbracket \} \\
\llbracket x \rrbracket + \llbracket y \rrbracket &= \{ \text{assumption that } \llbracket x \rrbracket = \llbracket x' \rrbracket \} \\
\llbracket x' \rrbracket + \llbracket y \rrbracket &= \{ \text{definition of } \llbracket \ ] \rrbracket \} \\
\llbracket \text{Add} \ x' \ y \rrbracket &
\end{align*}
\]

and

\[
\begin{align*}
\llbracket \text{Add} \ x \ y \rrbracket &= \{ \text{definition of } \llbracket \ ] \rrbracket \} \\
\llbracket x \rrbracket + \llbracket y \rrbracket &= \{ \text{assumption that } \llbracket y \rrbracket = \llbracket y' \rrbracket \} \\
\llbracket x \rrbracket + \llbracket y' \rrbracket &= \{ \text{definition of } \llbracket \ ] \rrbracket \} \\
\llbracket \text{Add} \ x \ y' \rrbracket &
\end{align*}
\]

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 conditions as above.
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:

\[
prop :: \text{Expr} \rightarrow \text{Bool} \\
prop x = \text{and} \left\{ \text{eval } x = \text{eval } x' \mid x' \leftarrow \text{trans } x \right\}
\]

If we now use the Quickcheck library to write a random generator for expressions, which is provided as part of the online supplementary Haskell code for the article, it can then be used to automatically test the property on a range of examples:

> quickCheck prop
>>> OK, passed 100 tests.

The idea of testing semantic properties in this lightweight manner has proved invaluable in our research work. For example, in our work on compiler verification, 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 Wright (2005) shows how the 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). Using QuickCheck to test semantic properties is explored in further detail in (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 from Section 3:

\[
eval :: \text{Expr} \rightarrow \text{Integer} \\
eval \text{ (Val } n \text{) } = n \\
eval \text{ (Add } x \text{ } y \text{)} = \text{eval } x + \text{eval } y
\]

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

Formally, an abstract machine for evaluating expressions of type \(\text{Expr}\) can be given by three components: a type \(\text{Cont}\) of control stacks, a function \(\text{eval} :: \text{Expr} \rightarrow \text{Cont} \rightarrow \text{Integer}\)
that evaluates an expression and then continues by executing the given control stack, and finally, a function $exec :: Cont \rightarrow Integer \rightarrow Integer$ that executes a control stack given the integer that resulted from evaluating an expression. The desired relationship between the component functions is captured by the following simple equation:

$$eval\ 'e\ c = exec\ c\ (eval\ e)$$

(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:

$$exec\ c\ (eval\ (Val\ n))$$
$$= \{\text{applying } eval\ \}$$
$$exec\ c\ n$$

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

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

That is, if the expression is an integer value it is already fully evaluated, and we simply execute the control stack using this integer as an argument. For the inductive case, $e = Add\ x\ y$, we begin in the same way as above by applying the evaluation function:

$$exec\ c\ (eval\ (Add\ x\ y))$$
$$= \{\text{applying } eval\ \}$$
$$exec\ c\ (eval\ x + eval\ y)$$

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:

$$exec\ c'\ (eval\ y) = exec\ c\ (eval\ x + eval\ y)$$
First of all, we generalise from the specific values \( \text{eval } x \) and \( \text{eval } y \) to give:

\[
\text{exec } c' \ m = \text{exec } c \ (n + m)
\]

Note that we can’t simply use this equation as a definition for \( \text{exec} \), 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 \( \text{Cont} \) type that takes these two variables as arguments,

\[
\text{ADD} :: \text{Integer} \to \text{Cont} \to \text{Cont}
\]

and defining a new equation for \( \text{exec} \) as follows:

\[
\text{exec} \ (\text{ADD } n \ c) \ m = \text{exec } c \ (n + m)
\]

That is, executing a control stack of the form \( \text{ADD } n \ c \) 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:

\[
\text{exec } c \ (\text{eval } x + \text{eval } y) \\
= \{ \text{define: } \text{exec} \ (\text{ADD } n \ c) \ m = \text{exec } c \ (n + m) \} \\
\text{exec} \ (\text{ADD } (\text{eval } x) \ c) \ (\text{eval } y) \\
= \{ \text{induction hypothesis for } y \} \\
\text{eval } y \ (\text{ADD } (\text{eval } x) \ c)
\]

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

\[
\text{exec } c' \ (\text{eval } x) = \text{eval } y \ (\text{ADD } (\text{eval } x) \ c)
\]

As with the case for \( y \), we first generalise from \( \text{eval } x \) to give

\[
\text{exec } c' \ n = \text{eval } y \ (\text{ADD } (\text{eval } x) \ c)
\]

and then package up the free variables \( y \) and \( c \) into the argument \( c' \) by adding a new constructor to \( \text{Cont} \) that takes these variables as arguments

\[
\text{EVAL} :: \text{Expr} \to \text{Cont} \to \text{Cont}
\]

and defining a new equation for \( \text{exec} \) as follows:

\[
\text{exec} \ (\text{EVAL } y \ c) \ n = \text{eval } y \ (\text{ADD } n \ c)
\]

That is, executing a control stack of the form \( \text{EVAL } y \ c \) given an integer argument \( n \) proceeds by evaluating the expression \( y \) and then executing the control stack \( \text{ADD } n \ c \).

Using these ideas, the calculation can now be completed:

\[
\text{eval } y \ (\text{ADD } (\text{eval } x) \ c) \\
= \{ \text{define: } \text{exec} \ (\text{EVAL } y \ c) \ n = \text{eval } y \ (\text{ADD } n \ c) \} \\
\text{exec} \ (\text{EVAL } y \ c) \ (\text{eval } x)
\]
The final term now has the required form (does not refer to \textit{eval}), from which we conclude that the following definition satisfies specification (1) in the inductive case:

\[
\text{eval}' \ (\text{Add} \ x \ y) \ c = \text{eval}' \ x \ (\text{EVAL} \ y \ c)
\]

That is, if the expression is an addition we proceed by evaluating the first argument expression \(x\), with the term \(\text{EVAL} \ y\) placed on the control stack to indicate that the second argument expression \(y\) should be evaluated once that of \(x\) has completed. In this manner, the definition makes explicit that addition is evaluated in left-to-right order.

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

\[
\text{eval} \ e
\]

\[
= \ { \text{define:} \ \text{exec} \ \text{HALT} \ n = n \}
\]

\[
\text{exec} \ \text{HALT} \ (\text{eval} \ e)
\]

\[
= \ { \text{specification (1)} \}
\]

\[
\text{eval}' \ e \ \text{HALT}
\]

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

\begin{verbatim}
data Cont = HALT | EVAL Expr Cont | ADD Integer Cont
eval :: Expr \to Integer
eval e = eval' e HALT

eval' :: Expr \to Cont \to Integer
eval' (Val n) c = exec c n

\end{verbatim}

\begin{verbatim}
eval' (Add x y) c = eval' x (EVAL y c)
\end{verbatim}

\begin{verbatim}
exec :: Cont \to Integer \to Integer
exec HALT n = n

\end{verbatim}

\begin{verbatim}
exec (EVAL y c) n = eval' y (ADD n c)
exec (ADD n c) m = exec c (n + m)
\end{verbatim}

Note that \textit{eval}' and \textit{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:
It's Easy As 1,2,3

\[
\begin{align*}
\text{eval} (\text{Add} (\text{Val} 1) (\text{Val} 2)) &= \text{eval}' (\text{Add} (\text{Val} 1) (\text{Val} 2)) \text{HALT} \\
&= \text{eval}' (\text{Val} 1) (\text{EVAL} (\text{Val} 2) \text{HALT}) \\
&= \text{exec} (\text{EVAL} (\text{Val} 2) \text{HALT}) 1 \\
&= \text{eval}' (\text{Val} 2) (\text{ADD} 1 \text{HALT}) \\
&= \text{exec} (\text{ADD} 1 \text{HALT}) 2 \\
&= \text{exec} \text{HALT} 3 \\
&= 3
\end{align*}
\]

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

**Further reading** The idea of deriving machines from semantics is due to Reynolds [1972], and was later popularised by Danvy and his collaborators (Ager et al., 2003), whose work also explores the connection between control stacks and evaluation contexts. This section is based upon (Hutton & Wright, 2006; Hutton & Bahr, 2016), which also show how to calculate machines for extended versions of our expression language, and how such calculations can be mechanically checked using the Coq proof assistant. Similar techniques can be used to calculate compilers for stack (Bahr & Hutton, 2015) and register machines (Hutton & Bahr, 2017; Bahr & Hutton, 2020), typed languages (Pickard & Hutton, 2021) and non-terminating languages (Bahr & Hutton, 2021).

### 9 Adding Exceptions

We have now seen how a range of semantic concepts can be presented using the language of integers and addition. For our final example, we consider how this language can be extended with other features of interest. In particular, we show how support for exceptions (Goodenough, 1975) can be added, and how the resulting language can be used as the basis for explaining and verifying how exceptions can be compiled.

We begin by extending our simple expression language with primitives for throwing and catching an exception, by adding two new constructors:

\[
\text{data Expr} = \text{Val Integer} | \text{Add Expr Expr} | \text{Throw} | \text{Catch Expr Expr}
\]

Informally, *Throw* abandons the current evaluation and throws an exception, while *Catch x h* behaves as the expression *x* unless it throws an exception, in which case the catch behaves as the ‘handler’ expression *h*. As with the basic language of integers and addition, this extended language does not provide features that are necessary for actual programming, but it does provide just enough structure to consider how they can be compiled.

To define the semantics for the extended language, we first recall the *Maybe* type:
type Maybe a = Nothing | Just a

That is, a value of type Maybe a is either Nothing, which we view as an exceptional value, or has the form Just x, which we view as a normal value (Spivey, 1990). Using this type, an evaluation function for our extended language can be defined as follows:

```haskell
eval :: Expr → Maybe Integer
eval (Val n) = Just n
eval (Add x y) = case eval x of
  Nothing → Nothing
  Just n → case eval y of
    Nothing → Nothing
    Just m → Just (n + m)
eval (Throw) = Nothing
 eval (Catch x h) = case eval x of
   Nothing → eval h
   Just n → Just n
```

Note that addition propagates an exception thrown in either argument, while catch only evaluates its second argument if the first throws an exception. Now let us consider compiling expressions into code for execution using a stack machine, where code comprises a list of operations on the stack, and a stack comprises a list of items:

```haskell
type Code = [Op]
type Stack = [Item]
```

For our expression language, the following set of basic operations suffice:

```haskell
data Op = PUSH Integer | ADD | THROW | MARK Code | UNMARK
```

Informally, PUSH pushes an integer value onto the stack, ADD replaces the top two values by their sum, THROW raises an exception, MARK pushes a piece of handler code onto the stack, and UNMARK removes such code from the stack. Because the stack can contain both integer values and handler code, there are two forms of stack items:

```haskell
data Item = VAL Integer | HAN Code
```

Using the above type declarations, it is now straightforward to define a function that executes code using an initial stack to give a final stack:

```haskell
exec :: Code → Stack → Stack
exec [] s = s
exec (PUSH n:c) s = exec c (VAL n:s)
exec (ADD:c) (VAL m:VAL n:s) = exec c (VAL (n + m):s)
exec (THROW:c) s = unwind s
exec (MARK h:c) s = exec c (HAN h:s)
exec (UNMARK:c) (x:HAN _:s) = exec c (x:s)
```
The cases for \texttt{PUSH} and \texttt{ADD} are as expected, with values on the stack tagged using \texttt{VAL}.

The case for \texttt{THROW} abandons the current execution and seeks a handler to deal with the exception, using an auxiliary function \texttt{unwind} that is defined as follows:

\begin{align*}
\texttt{unwind} &:: \text{Stack} \to \text{Stack} \\
\texttt{unwind} \; [\;] & = [\;] \\
\texttt{unwind} \; (\texttt{VAL} \; n \; s) & = \texttt{unwind} \; s \\
\texttt{unwind} \; (\texttt{HAN} \; h \; s) & = \texttt{exec} \; h \; s
\end{align*}

This function implements the idea of ‘stack unwinding’ \cite{Chase_1994a, Chase_1994b}, in which elements are popped from the stack until a handler is found, at which point execution then transfers to the handler code. Note that \texttt{exec} and \texttt{unwind} are defined mutually recursively, and correspond to two execution modes for the stack machine, the first when execution is proceeding normally, and the second when an exception has been thrown and a handler is being sought. If no handler is found then \texttt{unwind} returns the empty stack, which corresponds to the case when an exception is uncaught.

Returning to the remaining cases in the definition of \texttt{exec}, a \texttt{MARK} is executed simply by pushing the given handler code onto the stack, and dually, an \texttt{UNMARK} by popping this code from the stack. Between executing a mark and its corresponding unmark the code delimited by these two operations will have pushed its result value onto the stack, and hence when the handler is popped it will actually be the second-top item.

Now that we have implemented the stack machine, we can consider a compiler that translates expressions into code for the machine. We define the compiler in ‘accumulator style’ \cite{Wadler_1989} using an auxiliary function \texttt{comp} that takes an additional code parameter that is used to accumulate the compiled code, which is initially empty:

\begin{align*}
\texttt{compile} &:: \text{Expr} \to \text{Code} \\
\texttt{compile} \; e & = \texttt{comp} \; e \; [\;] \\
\texttt{comp} &:: \text{Expr} \to \text{Code} \to \text{Code} \\
\texttt{comp} \; (\texttt{Val} \; n) & = \texttt{PUSH} \; n \; : \texttt{c} \\
\texttt{comp} \; (\texttt{Add} \; x \; y) & = \texttt{comp} \; x \; (\texttt{comp} \; y \; (\texttt{ADD} : \texttt{c})) \\
\texttt{comp} \; (\texttt{Throw}) & = \texttt{THROW} : \texttt{c} \\
\texttt{comp} \; (\texttt{Catch} \; x \; h) & = \texttt{MARK} \; (\texttt{comp} \; h \; \texttt{c}) : \texttt{comp} \; x \; (\texttt{UNMARK} : \texttt{c})
\end{align*}

That is, \texttt{Val} \; n is compiled simply by pushing the value \; n onto the stack, while \texttt{Add} \; x \; y is compiled by first compiling the argument expressions \; x and \; y and then adding the resulting two values on top of the stack. In turn, \texttt{Throw} is compiled directly to the corresponding machine operation, while \texttt{Catch} \; x \; h is compiled by first marking the stack with the compiled code for the handler expression \; h, then compiling the expression to be evaluated \; x, and finally unmarking the stack by removing the handler. In this way, the mark and unmark operations delimit the scope of the handler \; h to the expression \; x, as the handler is only present on the stack during the execution of the expression.

Before moving on, there are a few further points to be noted about defining the compiler in accumulator style. First of all, the accumulating parameter can also be viewed as a ‘code continuation’, in the form of additional code to be executed after the code that results from compiling the expression argument. Secondly, it avoids the need to use the list append operator \((\,\,\,+)\) when defining the compiler, which improves efficiency as append
takes linear time in the length of its first argument. Finally, and most importantly here, it simplifies the proof of correctness of the compiler, which we will now consider.

The correctness of our compiler can now be captured by the following equation, which formalises the idea that compiling an expression preserves its meaning:

\[
\text{exec (compile } e \text{)} s = \begin{cases} 
\text{case eval } x \text{ of} & \\
\text{Nothing} & \rightarrow \text{unwind } s \\
\text{Just } n & \rightarrow \text{VAL } n \cdot s
\end{cases}
\]

That is, compiling an expression and then executing the resulting code gives the same final stack as evaluating the expression and then continuing in the appropriate way. In particular, if evaluation fails the stack is unwound to seek a handler to deal with the exception, and if evaluation succeeds the resulting value is simply pushed onto the stack.

Because `compile` is defined in terms of another function `comp`, not surprisingly the proof of the above result depends on the correctness of `comp`, which we specify using an auxiliary function `cont` that captures how to continue after evaluating the expression:

\[
\text{exec (comp } e \text{ c)} s = \text{cont (eval } e \text{) c s (3)}
\]

where

\[
\text{cont :: Maybe Integer } \rightarrow \text{ Code } \rightarrow \text{ Stack } \rightarrow \text{ Stack}
\]

\[
\text{cont Nothing c s = unwind s}
\]

\[
\text{cont (Just } n \text{) c s = exec } c \text{ (VAL } n : s) (3)
\]

The above specification for `comp` is essentially the same as the version for `compile`, except that in the failure case we also need to execute the additional code. The proof of (3) proceeds by structural induction on the expression `e`. In each case, we start with the left-hand side `exec (comp } e \text{ c)} s` and transform it by equational reasoning, aiming to end up with the right-hand side `cont (eval } e \text{) c s`. We verify the cases for throw and catch below; the cases for integers and addition proceed in a similar manner.

The base case for `e = \text{Throw}` can be verified simply by applying definitions:

\[
\text{exec (comp Throw } c \text{) s} = \begin{cases} 
\text{definition of comp } \} & \\
\text{exec (THROW : c) s} & \begin{cases} 
\text{definition of exec } \} & \\
\text{unwind s} & \begin{cases} 
\text{definition of cont } \} & \\
\text{cont Nothing c s} & \begin{cases} 
\text{definition of eval } \} & \\
\text{cont (eval Throw } c \text{) s}
\end{cases}
\end{cases}
\end{cases}
\]

The inductive case for `e = \text{Catch } x \text{ h}` is also straightforward, and as expected makes use of the induction hypotheses for the argument expressions `x` and `h`:

\[
\text{exec (comp (Catch } x \text{ y) c)} s = \begin{cases} 
\text{definition of comp } \} & \\
\text{exec (MARK (comp } h \text{ c) : comp s (UNMARK : c)) s} & \begin{cases} 
\text{definition of exec } \}
\end{cases}
\end{cases}
\]
exec \((comp\ x\ (UNMARK:\ c))\ (HAN\ (comp\ h\ c)\ :s)\)
= \{\ \text{induction hypothesis for}\ x\ \}\
cont\ (eval\ x)\ (UNMARK:\ c)\ (HAN\ (comp\ h\ c)\ :s)
= \{\ \text{definition of}\ cont\ \}\n
\text{case}\ eval\ x\ \text{of}
\text{Nothing}\ \rightarrow\ \text{unwind}\ (HAN\ (comp\ h\ c)\ :s)
\text{Just}\ n\ \rightarrow\ \text{exec}\ (UNMARK:\ c)\ (VAL\ n:\ HAN\ (comp\ h\ c)\ :s)
= \{\ \text{definition of}\ \text{unwind, exec}\ \}\n\text{case}\ eval\ x\ \text{of}
\text{Nothing}\ \rightarrow\ (\text{exec}\ (\text{comp}\ h\ c)\ s)
\text{Just}\ n\ \rightarrow\ (\text{exec}\ (\text{VAL}\ n:\ s))
= \{\ \text{induction hypothesis for}\ h\ \}\n\text{case}\ eval\ x\ \text{of}
\text{Nothing}\ \rightarrow\ (\text{cont}\ (\text{eval}\ h)\ c\ s)
\text{Just}\ n\ \rightarrow\ (\text{exec}\ (\text{VAL}\ n:\ s))
= \{\ \text{definition of}\ cont\ \}\n\text{case}\ eval\ x\ \text{of}
\text{Nothing}\ \rightarrow\ (\text{cont}\ (\text{eval}\ h)\ c\ s)
\text{Just}\ n\ \rightarrow\ (\text{Just}\ n)\ c\ s
= \{\ \text{distribution over case}\ \}\n\text{cont}\ (\text{case}\ eval\ x\ \text{of})
\text{Nothing}\ \rightarrow\ \text{eval}\ h
\text{Just}\ n\ \rightarrow\ (\text{Just}\ n)\ c\ s
= \{\ \text{definition of}\ \text{eval}\ \}\n\text{cont}\ (\text{eval}\ (\text{Catch}\ x\ h))\ c\ s

The ‘distribution over case’ step in the above proof relies on the fact that \text{cont} is strict in its first argument (\text{cont} \bot\ c\ s = \bot, where \bot represents an undefined value), which is indeed the case because \text{cont} is defined by pattern matching on this argument.

Finally, we return to the correctness equation \(\text{2}\) for our top-level compilation function \text{compile}, which can now be verified using the correctness of \text{comp}:

exec\ (\text{compile}\ e)\ s
= \{\ \text{definition of}\ \text{compile}\ \}\nexec\ (\text{comp}\ e\ [])\ s
= \{\ \text{correctness of}\ \text{comp}\ \text{[4]}\ \}\ncont\ (eval\ e\ [])\ s
= \{\ \text{definition of}\ cont\ \}\n\text{case}\ eval\ e\ \text{of}
\text{Nothing}\ \rightarrow\ \text{unwind}\ s
\text{Just}\ n\ \rightarrow\ \text{exec}\ [\] (\text{VAL}\ n:\ s)
= \{\ \text{definition of}\ \text{exec}\ \}\n\text{case}\ eval\ e\ \text{of}
\text{Nothing}\ \rightarrow\ \text{unwind}\ s
\text{Just}\ n\ \rightarrow\ \text{VAL}\ n:\ s
Exceptions have traditionally been viewed as an advanced topic in compilation, usually being discussed only briefly in courses, textbooks and research articles, and in many cases not at all. In this section we showed how the basic method of compiling exceptions using stack unwinding can be explained and verified using elementary techniques. Starting with the simple language of integers and addition was key to our development, which was the first time a compiler for exceptions had been proved correct.

Further reading This section is based upon (Hutton & Wright, 2004), which also shows how to verify a compiler for exceptions that pushes jump addresses for handlers onto the stack, rather than handler code itself. However, the compiler correctness proof presented here is simpler than the version in our original article, by virtue of writing the compiler in accumulator style. Our minimal language has also been used as the basis for verifying compilers for a range of other features, including interrupts (Hutton & Wright, 2007), transactions (Hu & Hutton, 2009) and concurrency (Hu & Hutton, 2010).

10 Summary and Conclusion

In this article we have show how a range of semantic concepts can be presented in a simple manner using the language of integers and addition. We have considered various semantic approaches, how induction principles can be used to reason about semantics, and how semantics can be implemented in different ways. 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, we have only scratched the surface of the subject of semantics here. For readers interested in learning more, there are many excellent textbooks such as (Harper, 2016; Pierce, 2002; Reynolds, 1998; Winskel, 1993), summer schools including the Oregon Programming Languages Summer School (OPLSS, 2021) and the Midlands Graduate School (MGS, 2021), and numerous online lecture notes and tutorials. We hope that our simple language provides others with a useful gateway and tool for exploring further aspects of programming language semantics. In this setting, it’s easy as 1,2,3.

Acknowledgements

I would like to thanks Jeremy Gibbons, Ralf Hinze and the JFP reviewers for many useful comments that significantly improved the article. This work was funded by EPSRC grant EP/P00587X/1, Unified Reasoning About Program Correctness and Efficiency.

Conflicts of Interest

None.

References

It's Easy As 1,2,3


Graham Hutton

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


