
Access from the University of Nottingham repository:
http://eprints.nottingham.ac.uk/230/1/semantics.pdf

Copyright and reuse:

The Nottingham ePrints service makes this work by researchers of the University of Nottingham available open access under the following conditions.

This article is made available under the University of Nottingham End User licence and may be reused according to the conditions of the licence. For more details see:
http://eprints.nottingham.ac.uk/end_user_agreement.pdf

A note on versions:

The version presented here may differ from the published version or from the version of record. If you wish to cite this item you are advised to consult the publisher's version. Please see the repository url above for details on accessing the published version and note that access may require a subscription.

For more information, please contact eprints@nottingham.ac.uk
Fold and Unfold for Program Semantics

Graham Hutton
Languages and Programming Group
Department of Computer Science
University of Nottingham, UK

http://www.cs.nott.ac.uk/~gmh

Abstract

In this paper we explain how recursion operators can be used to structure and reason about program semantics within a functional language. In particular, we show how the recursion operator fold can be used to structure denotational semantics, how the dual recursion operator unfold can be used to structure operational semantics, and how algebraic properties of these operators can be used to reason about program semantics. The techniques are explained with the aid of two main examples, the first concerning arithmetic expressions, and the second concerning Milner’s concurrent language CCS. The aim of the paper is to give functional programmers new insights into recursion operators, program semantics, and the relationships between them.

1 Introduction

Many computations are naturally expressed as recursive programs defined in terms of themselves, and properties proved of such programs using some form of inductive argument. Not surprisingly, many programs will have a similar recursive structure, and many proofs will have a similar inductive structure. To avoid repeating the same patterns of program and proof again and again, special recursion operators and proof principles that abstract out the common patterns can be introduced, allowing us to concentrate on the details that are specific to each different application.

In the functional programming community, much previous work in this area has focussed on a recursion operator called fold, and on its associated proof principle called universality. Fold captures a common programming pattern in which a list of values is processed in a certain recursive manner, and universality captures a common pattern of inductive proof concerning programs that process lists. Fold and universality have proved useful in a variety of application areas, including algorithm construction [1, 11, 2], hardware construction [7, 6], compiler construction [12], and automatic program transformation [20, 3, 8]. Using ideas from category theory, fold has been uniformly generalised from lists to a large class of recursive datatypes [10, 14].

In this paper we are concerned with the application of recursion operators in the area of program semantics. One of the most popular styles of semantics is the denotational approach [19], in which the meaning of programs is defined using a valuation function that maps programs into values in an appropriate semantic domain. The valuation function is defined using a set of recursion equations, and must be compositional in the sense that the meaning of a program is defined purely in terms of the meaning of its syntactic subcomponents. In fact, the pattern of recursion required by compositionality is precisely the pattern of recursion captured by fold. Hence, a denotational semantics can be characterised as a semantics defined by folding over program syntax. Although widely known in certain circles, many functional programmers are still not aware of this connection.

The recursion operator fold has a natural dual, called unfold, which captures a common programming pattern in which a list of values is produced (as opposed to processed) in a certain recursive manner. The dual proof principle, again called universality, captures a common pattern of coinductive proof concerning programs that produce lists. Unfold has also been generalised from lists to a large class of recursive datatypes [10, 14]. While applications of fold abound, relatively little attention has been given to unfold in the functional programming community.

Another popular style of semantics is the operational approach [17], in which the meaning of programs is defined using a transition relation that captures single execution steps in an appropriate abstract machine. The transition relation is defined using a set of inference rules, and the meaning of a program is given by repeatedly applying the relation to generate a transition tree that captures all possible execution paths of the program. In fact, the pattern of recursion used to construct transition trees is precisely the pattern of recursion captured by unfold. Hence, an operational semantics can be characterised as a semantics defined by unfolking to transition trees. This connection has been developed using category theory [18, 21], but most functional programmers are not aware of this connection.

In this paper we explain how recursion operators can be used to structure and reason about program semantics within the functional language Haskell [16]. In particular, we show how fold can be used to structure denotational semantics, how unfold can be used to structure operational semantics, and how algebraic properties of these operators can be used to reason about program semantics.

The techniques are explained with the aid of two main examples, the first concerning arithmetic expressions, and the second concerning Milner’s concurrent language CCS [15].
As the paper proceeds we adopt an increasingly categorical approach to semantics, to give a deeper understanding of the issues. However, previous knowledge of category theory is not required. The aim of the paper is to give functional programmers new insights into recursion operators, program semantics, and the relationships between them.

2 Denotational semantics

In denotational semantics [19], the meaning of terms is defined using a valuation function that maps terms into values in an appropriate semantic domain. In this section we explain how a denotational semantics can be characterised as a semantics defined by folding over syntactic terms.

Formally, a denotational semantics for a language $T$ of syntactic terms comprises two components: a set $V$ of semantic values, and a valuation function $\llbracket \cdot \rrbracket : T \rightarrow V$ that maps terms to their meaning as values. 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 $T$-subterms. When the set of semantic values is clear, a denotational semantics is often identified with a compositional valuation function.

2.1 Arithmetic expressions

As an example, let us consider a language of simple arithmetic expressions, built up from the set $\mathbb{Z}$ of integer values using the addition operator $\cdot$. The language $E$ of such expressions is defined by the following grammar:

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

We assume that parentheses can be used to disambiguate expressions if required. The grammar for expressions can be directly translated into a Haskell datatype definition, parameterised over the type of values for flexibility:

```haskell
data Expr = Val a | Add (Expr a) (Expr a)
```

For example, the expression $1 + (2 + 3)$ is represented by the value $\text{Add} (\text{Val} 1) (\text{Add} (\text{Val} 2) (\text{Val} 3))$. From now on, we mainly consider expressions represented in Haskell.

Arithmetic expressions have an obvious denotational semantics, given by taking $V$ as the Haskell type $\text{Int}$ of integers and $\llbracket \cdot \rrbracket : \text{Expr} \rightarrow \text{Int}$ as the evaluation function for expressions defined recursively as follows:

$$\llbracket \text{Val} n \rrbracket = n$$
$$\llbracket \text{Add} x y \rrbracket = [x] + [y]$$

This definition satisfies the compositionality requirement, because the meaning of compound expressions of the form $\text{Add} x y$ is defined purely by applying $+$ to the meanings of the subexpressions $x$ and $y$. The evaluation function can be translated directly into a Haskell function definition:

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

For example, $\text{eval} (\text{Add} (\text{Val} 1) (\text{Add} (\text{Val} 2) (\text{Val} 3))) = 1 + (2 + 3) = 6$, or drawing expressions as trees:

```
  +
  /|
/  |
/   |
```

Looking at this example, we see that an expression is evaluated by removing each constructor $\text{Val}$ (or equivalently, replacing each constructor $\text{Val}$ by the identity function id on integers), and replacing each constructor $\text{Add}$ by the addition function ($+$) on integers. That is, even though eval was defined recursively, its behaviour can be understood non-recursively as simply replacing the two constructors for expressions by the functions id and ($+$).

2.2 Fold for expressions

Abstracting from the specific case of eval, we can consider the general case of a denotational semantics $\text{deno}$ that gives meaning to arithmetic expression by replacing each $\text{Val}$ by a function $f$, and each $\text{Add}$ by a function $g$. By definition, a semantics defined in this manner will be compositional, because the meaning of addition is defined purely by applying $g$ to the meanings of the two argument expressions:

$$\llbracket \text{Val} n \rrbracket = f n$$
$$\llbracket \text{Add} x y \rrbracket = g (\llbracket x \rrbracket) (\llbracket y \rrbracket)$$

Since the behaviour of such functions can be understood non-recursively, why don’t we actually define them in this manner? This is precisely what fold allows us to do. Using fold for arithmetic expressions, we can define denotational semantics for expressions simply by supplying the function $f$ that replaces each $\text{Val}$ and the function $g$ that replaces each $\text{Add}$. For example, using fold the denotational semantics eval can be simply defined as follows:

```haskell
eval = fold id ($+$)
```

As another example, using fold we can define an alternative semantics comp that doesn’t evaluate expressions directly, but rather compiles expressions into a list of instructions for execution using a stack. As for eval, defining the semantics using fold makes it compositional by definition:

```haskell
data Inst = PUSH Int | ADD

comp :: Expr Int -> [Inst]
comp = fold f g

where

  \begin{align*}
  f n &= [\text{PUSH} n] \\
  g x y &=[\text{ADD}] \\
  g &\hspace{1em} x+y \\
  \end{align*}
```

```bash
For example, \text{comp} (\text{Add} (\text{Val} 1) (\text{Add} (\text{Val} 2) (\text{Val} 3))) = [\text{PUSH} 1, \text{PUSH} 2, \text{PUSH} 3, \text{ADD}, \text{ADD}].

The fold function itself can be defined simply by abstracting on the free variables $f$ and $g$ in the general definition of a denotational semantics $\text{deno}$ for expressions:

```haskell
fold f g (Val n) = f n
fold f g (Add x y) = g (fold f g x) (fold f g y)
```
The type of fold is given by the following inference rule:

\[
f :: a \rightarrow b \quad g :: b \rightarrow b \rightarrow b
\]

\[
fold f g :: Expr a \rightarrow b
\]

2.3 Generalising

Of course, the use of fold to define denotational semantics is not specific to our language of arithmetic expressions, but can be generalised to many other languages. For example, consider extending expressions with integer variables of the form \(Var c\) for any character \(c\). Then the fold operator would simply be generalised to take an extra argument function \(h\) to replace each constructor \(Var\) in an expression:

\[
fold f g h (Val n) = f n
\]

\[
fold f g h (Add x y) = g (fold f g h x)
\]

\[
fold f g h (Var c) = h c
\]

In turn, the denotational semantics \(eval\) would be generalised to give the meaning of expressions as functions from stores (containing the value of each variable) to integers. Assuming a type \(Store\) for stores and a function \(find\) for looking up the value of a variable in a store, \(eval\) is defined using the generalised fold as follows:

\[
eval :: Expr \rightarrow (Store \rightarrow Int)
\]

\[
eval = fold f g h
\]

where

\[
f f n = \lambda s \rightarrow n
\]

\[
g fx fy = \lambda s \rightarrow fx s + fy s
\]

\[
h c = \lambda s \rightarrow find c s
\]

Again, defining the semantics using fold makes it compositional by definition. As in this example, the set of semantic values for most non-trivial languages will usually involve functions in some way, for a more general discussion on the use of fold to return functions, see [4].

In general, we have the following simpk connection between denotational semantics and fold operators:

\[
\begin{array}{c}
\text{Denotational semantics} \\
\text{fold}
\end{array}
\]

3 Operational semantics

In operational semantics [17], the meaning of terms is defined using a transition relation that captures execution steps in an appropriate abstract machine. In this section we explain how an operational semantics can be characterised as a semantics defined by unfolding to transition trees.

Formally, an 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 states to all the states that can be reached by performing a single execution step. (For some applications, a more general notion of transition relation may be appropriate, 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 usually write this as \(s \rightarrow s'\). When the set of states is clear, an operational semantics is often identified with a transition relation.

3.1 Arithmetic expressions

Returning to our example from the previous section, simple arithmetic expressions have an obvious operational semantics, given by taking \(S\) as the Haskell type \(Expr\) of expressions, and \(\rightarrow \subseteq Expr \times Expr\) as the transition relation defined by the following three inference rules:

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

The first rule states that two values can be added together to give a single value, and the last two rules permit the first rule to be applied to either argument of an addition expression. For example, the (concrete) expression \((1 + 2) + (3 + 4)\) has two possible transitions, because the first transition rule can be applied to either argument of the top-level addition:

\[
(1 + 2) + (3 + 4) \rightarrow 3 + (3 + 4) \\
(1 + 2) + (3 + 4) \rightarrow (1 + 2) + 7
\]

By repeated application of a transition relation, it is possible to generate a transition tree that captures all possible execution paths for a syntactic term. For example, the expression \((1 + 2) + (3 + 4)\) gives rise to the following transition tree, which captures the two possible execution paths:

\[
\begin{align*}
1 + 2 + (3 + 4) & \rightarrow 3 + (3 + 4) \\
3 + (3 + 4) & \rightarrow 3 + 7 \\
3 + 7 & \rightarrow 10 \\
\end{align*}
\]

The inference rules defining the transition relation for expressions can be easily translated into a Haskell function definition. The relation is represented as a list-valued function that maps expressions to lists of expressions that can be reached by performing a single execution step:

\[
\begin{align*}
\text{trans} :: Expr \rightarrow [Expr]
\end{align*}
\]

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

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:

\[
\text{data Tree a = Node a [Tree a]}
\]

\[
\begin{align*}
\text{exec} :: Expr \rightarrow Tree (Expr Int) \\
\text{exec} e & = \text{Node e [exec e' | e' \leftarrow \text{trans} e]}
\end{align*}
\]

Looking at the definition of exec, we see that an expression is executed to yield a tree by taking the expression unchanged as the root of the tree (or equivalently, applying the identity function \(id\) on expressions), and generating a
list of residual expressions to be processed to give the sub-
trees by applying the trans function. That is, even though exec was defined recursively, its behaviour can be under-
stood non-recursively as simply applying the identity func-
tion id to generate the root expression, and the transition 
function trans to generate a list of residual expressions to 
be processed to generate the subtrees.

3.2 Unfold for trees

Abstracting from the specific case of exec, we can consider 
the general case of an operational semantics oper that gives 
meaning as trees by using a function f to generate the root of 
the tree, and a function g to generate a list of residual 
values to be processed to generate the subtrees:

\[
\text{oper} :: a \rightarrow \text{Tree } b
\]
\[
\text{oper } x = \text{Node } (f x) \left[ \text{oper } x' \mid x' \leftarrow g x \right]
\]

Since the behaviour of such functions can be understood 
non-recursively, why don't we actually define them in this 
manner? This is precisely what unfold allows us to do. 
Using unfold for trees, we can define operational semantics 
as trees simply by supplying the function f that generates 
the root of the tree, and the function g that generates the 
residual values. For example, using unfold the operational 
semantics exec can be simply defined as follows:

\[
\text{exec} = \text{unfold } \text{id } \text{trans}
\]

The unfold function itself is defined simply by abstracting 
on the free variables f and g in the general definition of 
an operational semantics oper as trees:

\[
\text{unfold } f g x = 
\text{Node } (f x) \left[ \text{unfold } f g x' \mid x' \leftarrow g x \right]
\]

The type of unfold is given by the following inference rule:

\[
f :: a \rightarrow b \\
g :: a \rightarrow [a] \\
\text{unfold } f g :: a \rightarrow \text{Tree } b
\]

3.3 Generalising

Of course, the use of unfold to define operational semantics 
is not specific to our language of arithmetic expressions, but 
can be generalised to many other languages. That is, we 
have the following simple connection between operational 
semantics and unfold operators:

<table>
<thead>
<tr>
<th>Operational semantics</th>
<th>unfold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unfolding to transition trees</td>
<td></td>
</tr>
</tbody>
</table>

This is precisely dual to the connection for denotational se-
manitics given in the previous section. Hence, taking a struc-
tured approach to program semantics using recursion opera-
tors has revealed a duality between denotational and oper-
tional semantics that might otherwise have been missed. In 
the next section we will see that using recursion operators 
also brings benefits when proving properties of semantics.

4 Reasoning about semantics

One of the main reasons for defining the formal semantics 
of programming languages is to support formal reasoning 
about languages and programs written in them. In this sec-
tion we explain how properties of semantics can be proved 
using the universality of the recursion operator fold[13, 14], 
rather than explicit structural induction.

4.1 Arithmetic expressions

Consider the following three equations concerning our se-
manitics for (finite) simple arithmetic expressions:

\[
(1) \quad \text{and } [\text{deno } e' = \text{deno } e \mid e' \leftarrow \text{trans } e] = \text{True}
\]
\[
(2) \quad \text{and } [\text{size } e' < \text{size } e \mid e' \leftarrow \text{trans } e] = \text{True}
\]
\[
(3) \quad \text{and } [n = \text{deno } e \mid n \leftarrow \text{vals } (\text{oper } e)] = \text{True}
\]

The first equation states that the transition function pre-
serves the denotational semantics of expressions. The sec-
ond equation states that the transition function decreases 
the size of expressions, where the size is defined as the num-
ber of add constructors. The last equation states that the 
denotational and operational semantics are equivalent, in 
the sense that the integer values in the transition tree gener-
ated by the operational semantics are all equal to the value 
obtained from the denotational semantics. The auxiliary 
functions size and vals are easy to define.

Equations (1) and (2) above can be proved by induction 
on the structure of e. In turn, by making use of these two 
auxiliary results, (3) can be proved by induction on the size 
of e. However, the two proofs using structural induction can 
also be proved using the universality of fold, which avoids 
the need for explicit use of induction.

4.2 Universality for expressions

For simple arithmetic expressions, the universality of fold 
is captured by the following equivalence:

\[
\text{h } (\text{Val } n) = f n \\
\text{h } (\text{Add } x y) = g (h x) (h y)
\]

This equivalence states that fold f g is the unique solution 
to the first two equations, and can itself be proved using 
a simple structural induction. Indeed, the two equations 
are precisely the assumptions required to show that h = 
fold f g using structural induction. For specific cases then, 
by verifying the two assumptions (which can typically be 
done without the need for induction), we can then appeal 
to universality to complete the inductive proof that h = 
fold f g. In this manner, universality captures a common 
pattern of inductive proof, just as fold itself captures a 
common pattern of recursive definition.

To prove equation (1) above using the universality of 
fold, it must first be expressed in the form h = fold f g. 
In this case, h can be defined simply by abstracting over e 
on the left-hand side of the equation:

\[
h e = [\text{deno } e = \text{deno } e' \mid e' \leftarrow \text{trans } e]
\]
Hence, by appealing to the universality of which the nodes represent the states of the process, and the can be described by a (possibly infinite) transition tree, in continue as following two equations:

\[
\begin{align*}
    h(\text{Val } n) &= \text{True} \\
    h(\text{Add } x \ y) &= (h \ x) && (h \ y)
\end{align*}
\]

These equations can now be verified by routine calculations, without the need for an explicit induction. Universality can also be used to prove (2) in a similar way, again without the need for an explicit induction.

5 Concurrent processes in CCS

Up to this point, all our examples have been concerned with arithmetic expressions. For the remainder of the paper we show how our techniques apply to a real-life example, Milner's language CCS (Calculus of Concurrent Systems) for describing concurrent processes [15]. In this section we consider the Haskell datatypes required for the syntax and semantics of CCS processes, and show how they can be defined in an abstract manner as least fixpoints of functors. As we shall see in subsequent sections, this approach will permit a more abstract treatment of the semantics of processes.

Given a set \( N \) of process names, and a set \( A \) of process actions, the language \( P \) of processes in CCS is defined by the following grammar:

\[
P :::= N \quad \text{\textup{constants}} \\
    a.P \quad \text{\textup{prefixing}} \\
    \sum_i P_i \quad \text{\textup{parallelism}} \\
    P \| P' \quad \text{\textup{choice}} \\
    P[A] \quad \text{\textup{restriction}} \\
    P[f] \quad \text{\textup{labelling}}
\]

We assume that parentheses can be used to disambiguate processes if required. Named processes are defined by (possibly recursive) equations. The set \( A \) of actions is assumed to comprise input actions \( a, b, c, \ldots \), the corresponding output actions \( a, b, c, \ldots \), and the silent action \( \tau \) used to indicate synchronisation. A relabelling function \( f \) is a function from actions to actions that preserves their underlying structure, in the sense that \( f(\tau) = f(\tau) \) and \( f(\tau) = \tau \).

As a simple example of a process, consider the recursive equation \( A = a.A + b.A \). Intuitively, this equation defines the process \( A \) that can either perform the action \( a \) and then continue as \( A \) again, or perform the action \( b \) and then continue as \( A \) again. More formally, the meaning of a process can be described by a (possibly infinite) transition tree, in which the nodes represent the states of the process, and the edges are labelled with the actions that are performed in moving between states. For example, the meaning of \( A \) is given by the infinite tree pictured in Figure 1.

Assuming types \textup{Name} and \textup{Act} for names and actions respectively, the grammar for processes can be directly translated into a Haskell datatype definition:

\[
data \text{Proc} = \text{Con Name} \\
  \mid \text{Pre Act Proc} \\
  \mid \text{Cho [Proc]} \\
  \mid \text{Par Proc Proc} \\
  \mid \text{Res Proc Act} \\
  \mid \text{Rel Proc (Act \rightarrow Act)}
\]

(\textit{The Proc type could be parameterised over the type of actions, but we use a fixed type \textup{Act} for simplicity.}) In turn, a datatype for trees can be defined as follows:

\[
data T t = \text{Node ([Act, Tree]}\]

However, there is another approach to defining Proc and Tree that will permit the semantics of processes as trees to be defined in a more abstract manner. Rather than defining these types directly as recursive datatypes, we prefer to define them indirectly as least fixpoints of functors.

5.1 Least fixpoints

In semantics, it is common to model recursively defined values as least fixpoints of non-recursively defined functions [19]. For the special case of recursively defined types, the least fixpoint \textup{Fix} \( f \) of a type constructor \( f \) (a function from types to types) can be defined in Haskell as follows:

\[
data \text{type Fix} f = \text{In} (f \ \text{Fix} f)
\]

For example, the recursive type \text{Proc} can be expressed as the least fixpoint of a non-recursive type constructor \( P \), where the definition for \( P \) is precisely the same as for the original Proc type, except that each recursive call within the definition is replaced by an instance of a type parameter \( p \):

\[
data \text{type Proc} = \text{Fix } P
\]

\[
data \text{P p} = \text{Con Name} \\
  \mid \text{Pre Act p} \\
  \mid \text{Cho [p]} \\
  \mid \text{Par p p} \\
  \mid \text{Res p Act} \\
  \mid \text{Rel p (Act \rightarrow Act)}
\]

Constructors for the new Proc type are defined simply by applying the tag In to the constructors for \( P \):

\[
data \text{con n} = \text{In} \ (\text{Con n}) \\
  \text{pre a p} = \text{In} \ (\text{Pre a p}) \\
  \text{cho ps} = \text{In} \ (\text{Cho ps}) \\
  \text{par p q} = \text{In} \ (\text{Par p q}) \\
  \text{res p a} = \text{In} \ (\text{Res p a}) \\
  \text{rel p f} = \text{In} \ (\text{Rel p f})
\]

In turn, the recursive type Tree can be expressed as the least fixpoint of a non-recursive type constructor \( T \):

\[
data \text{type Tree} = \text{Fix } T
\]

\[
data \text{T t} = \text{Node ([Act,t]}\]
Given the above definitions, it can be shown that the Proc and Tree types defined as least fixpoints are isomorphic to the original types defined using explicit recursion. That is, the types are equivalent in the sense that there is a one-to-one correspondence between their values.

### 5.2 Functors

The next concept to be considered is that of a functor, which comes from category theory \[9\]. The notion of a functor is captured as a built-in class in Haskell, defined as follows:

```haskell
class Functor f where
  map :: (a -> b) -> (f a -> f b)

```

This definition states that a type constructor \( f \) is a member of the class Functor if it is equipped with a map function that lifts functions of type \( a \to b \) to functions of type \( f a \to f b \). Although not made explicit in the Haskell definition, a functor must also preserve the identity function and distribute over function composition, in the sense that:

```
map id = id
map (g . h) = (map g) . (map h)
```

For example, the type constructor \( P \) can be made into an instance of the class Functor with the following definition:

```haskell
instance Functor P where
  map f x = case x of
    Con n -> Con n
    Pre a p -> Pre a (f p)
    Cho p q -> Cho [f p | p <- ps]
    Par p q -> Par (f p) (f q)
    Res p a -> Res (f p) a
    Rel p g -> Rel (f p) g
```

It is easy to verify that this definition satisfies the equations required of a functor. In turn, the type constructor \( T \) can be made into an instance of the class Functor as follows:

```haskell
instance Functor T where
  map f (Node xs) = Node [(a, f t) | (a, t) <- xs]
```

In summary, we have now expressed the recursive type Proc as the least fixpoint of a non-recursive functor \( P \), and the recursive type Tree as the least fixpoint of the non-recursive functor \( T \). The map functions for both functors play no role yet, but they will in subsequent sections.

### 6 Operational semantics of CCS

As for most languages involving some form of concurrency, the standard semantics for CCS is an operational semantics \[12\]. In this section we show how the operational semantics for processes as trees can be defined in Haskell in an abstract manner using a polytypic version of unfold.

The operational semantics of CCS is given by a transition relation \( - \subseteq P \times \alpha \times P \), where \( P \) is the set of processes, and \( \alpha \) is the set of actions. If \( (P, a, P') \in - \), we say that the process \( P \) can perform the action \( a \) to become the process \( P' \), and usually write this as \( P \leadsto P' \). The transition relation \( - \) is defined by the following set of inference rules:

\[
\begin{align*}
P \triangleright a \triangleright P' &\quad (A = P) \\
A \triangleright P &\quad (a, P \triangleright P)
\end{align*}
\]

Here, \( P \triangleright P' \) is the set of actions. If \( (P, a, P') \in - \), we say that the process \( P \) can perform the action \( a \) to become the process \( P' \), and usually write this as \( P \leadsto P' \). The transition relation \( - \) is defined by the following set of inference rules:

\[
\begin{align*}
P \triangleright a \triangleright P' &\quad (A = P) \\
A \triangleright P &\quad (a, P \triangleright P)
\end{align*}
\]

For example, the rules can be made in to an instance of the class Functor as follows:

```
instance Functor T where
  map f x = case x of
    Con n -> Con n
    Pre a p -> Pre a (f p)
    Cho p q -> Cho [f p | p <- ps]
    Par p q -> Par (f p) (f q)
    Res p a -> Res (f p) a
    Rel p g -> Rel (f p) g
```

By repeated application of the transition relation, it is possible to generate a (possibly infinite) transition tree that captures all possible execution paths for a process. For example, the process \( A \) gives rise to the tree in Figure 1.

The inference rules defining the transition relation for processes can be easily translated into a Haskell function definition. The relation is represented as a list-valued function that maps processes to lists of (action,process) pairs that arise from single execution steps:

```
trans :: Proc -> [(Act, Proc)]
trans (In x) = case x of
  Con n -> trans (defn n)
  Pre a p -> [(a, p)]
  Cho ps -> concat (map trans ps)
  Par p q -> [(a, par p q) | (a, p) <- trans p, (b, q) <- trans q]
  Res p a -> [(b, res p a) | (a, b) <- trans a]
  Rel p g -> [(f a, rel p g) | (f, g) <- trans p]
```

The auxiliary function defn maps process names to their definitions and should be defined as appropriate by the user, while synch decides if two actions can synchronise, and strip removes any bars from an action to give its underlying name. Both synch and strip are easy to define.

In turn, we can define an execution function that converts processes into trees by repeated application of the transition function using the unfold function for trees:

```
exec :: Proc -> Tree
exec = unfold trans
```

The general purpose unfold function for our type Tree of transition trees can itself be defined as follows:

```
unfold f x =
  in (Node [(a, unfold f x') | (a, x') <- f x])
```

However, by exploiting the fact that Tree is defined as the least fixpoint of a functor, the execution function can be defined in a more abstract manner by repeated application of a transition co-algebra using a polytypic version of unfold that is not specific to any particular recursive datatype.
6.1 Co-algebras

The concept of a co-algebra that we use comes from category theory, and generalises the idea of a transition function. In Haskell, a co-algebra for a functor \( f \) is a function of type

\[
a \to f \ a
\]

for some specific type \( a \). For example, the transition function for processes can be converted into a transition co-algebra for the functor \( T \) by the following simple definition:

\[
\text{trans'} :: \text{Proc} \to T \text{Proc} \\
\text{trans'} p = \text{Node} (\text{trans} p)
\]

A more general example of a co-algebra concerns the fixpoint for \( \text{Fix} f \). In particular, the inverse function out of the tag \( \text{In} \) is a co-algebra for any functor \( f \):

\[
\text{out} :: \text{Fix} f \to f (\text{Fix} f) \\
\text{out} (\text{In} x) = x
\]

6.2 Polymorphic unfold

The co-algebra \( \text{out} \) is special among all co-algebras for a functor \( f \), being in fact the final co-algebra. Technically, this means that for any other co-algebra \( g :: a \to f \ a \), there is a unique function \( \text{unfold} g :: a \to \text{Fix} f \) such that the following diagram commutes [13, 14]:

\[
\begin{array}{ccc}
a & \xrightarrow{\text{unfold} g} & \text{Fix} f \\
\downarrow & & \downarrow\text{out} \\
f a & \xrightarrow{\text{map} (\text{unfold} g)} & f (\text{Fix} f)
\end{array}
\]

Using this diagram and the fact that out is the inverse to \( \text{In} \), the \( \text{unfold} \) function itself can be defined as follows:

\[
\text{unfold} g = \text{In} . \text{map} (\text{unfold} g) . g
\]

That is, the function \( \text{unfold} g \) first applies the co-algebra \( g \) to break down an argument of type \( a \) into a structured value of type \( f \ a \), then applies the function \( \text{map} (\text{unfold} g) \) to recursively process each of the \( a \) components to give a value of type \( f \) (\( \text{Fix} f \)), and finally applies the tag \( \text{In} \) to give a value of the recursive type \( \text{Fix} f \). In this manner, \( \text{unfold} \) is a general purpose function for producing values of a recursive type using a simple pattern of recursion.

While previously we defined \( \text{unfold} \) functions that were specific to particular recursive datatypes (for example, trees), the above version of \( \text{unfold} \) is \textit{polymorphic} [5], in the sense that it can be used with any recursive datatype that can be expressed as the least fixpoint of a functor.

In the case of processes, because the datatype \( \text{Tree} \) is expressed as the least fixpoint of the functor \( T \), and the transition function is expressed as a co-algebra \( \text{trans'} \) for \( T \), the execution function that maps processes to trees can now be defined using the polymorphic version of \( \text{unfold} \):

\[
\text{exec :: Proc} \to \text{Tree} \\
\text{exec} = \text{unfold trans'}
\]

In summary, we have now expressed the operational semantics of processes as trees as the unique function \( \text{unfold} \) that makes the following diagram commute:

\[
\begin{array}{c}
\text{Proc} \xrightarrow{-} \text{unfold trans'} \\
\downarrow \text{trans'} \\
T \text{Proc} \xrightarrow{\text{map} (\text{unfold trans'})} T \text{Tree}
\end{array}
\]

7 Denotational semantics of CCS

In the previous section we defined an operational semantics for processes as trees by unfolding a transition function expressed as a co-algebra. In this section we consider the less well-known denotational semantics for processes as trees, and show how it can be defined in a dual manner by folding a combining function expressed as an \textit{algebra}.

7.1 Algebras

In the spirit of category theory, the notion of a co-algebra is dual to that of an algebra. In Haskell, an algebra for a functor \( f \) is a function of type

\[
f a \to a
\]

for some specific type \( a \). For example, the tag function \( \text{In} :: f (\text{Fix} f) \to \text{Fix} f \) is an algebra for any functor \( f \). A more specific example of a co-algebra concerns the semantics of processes as trees. In particular, it is natural to define an algebra for the functor \( f \) as follows:

\[
\text{comb :: T Tree} \to \text{Tree} \\
\text{comb x} = \text{In} . (\text{Node} \circ \text{case} x) \\
\text{Con n} \to \text{denode (eval (defn n))} \\
\text{Pre a t} \to [(\text{a}, \text{t})] \\
\text{Cho ts} \to \text{concat (map denode ts)} \\
\text{Par t u} \to [(\text{a}, \text{comb (Par t u'})] \\
\text{(a}, \text{t'}) \to \text{denode t} \\
\text{(b}, \text{u'}) \to \text{denode u} \\
\text{(a}, \text{t'}) \to \text{denode t'} \\
\text{(b}, \text{u'}) \to \text{denode u'} \\
\text{Res t a} \to [(\text{a}, \text{comb (Res t' a))}] \\
\text{(a}, \text{t'}) \to \text{denode t'} \\
\text{(b}, \text{u'}) \to \text{denode u'} \\
\text{Rel t f} \to [(\text{a}, \text{comb (Rel t f'))}] \\
\text{(a}, \text{t'}) \to \text{denode t'}
\]

The auxiliary function \( \text{eval} \) will be defined shortly, while \( \text{denode} \) is the destructor function for trees:

\[
\text{denode :: Tree} \to [(\text{Act.Tree})] \\
\text{denode (In (Node xs))} = \text{xs}
\]

We refer to \( \text{comb} \) as a \textit{combining} function, because it takes a value built by applying a CCS operator to trees rather than to processes, and combines the trees into a single tree by interpreting the operator in the appropriate manner for trees. For example, the third case for parallel composition \( \text{Par t u} \) states that if the tree \( t \) has an \( \text{a}-\text{labelled} \) branch to a subtree \( t' \), the tree \( u \) has a \( \text{b}-\text{labelled} \) branch to a subtree \( u' \), and the actions \( a \) and \( b \) can synchronise, then the resulting combined tree has a \( \text{a}-\text{labelled} \) branch to the recursively computed subtree \( \text{comb} (\text{Par t' u'}) \).
7.2 Polytypic fold

The algebra $\text{In}$ is special among all algebras for a functor $f$, being in fact the initial algebra. Technically, this means that for any other algebra $g :: f a \to a$, there is a unique strict function $\text{fold} \ g :: \text{Fix} \ f \to a$ such that the following diagram commutes [13, 14]:

\[
\begin{array}{c}
\text{In} \\
\downarrow \\
\text{fold} \ g
\end{array}
\quad
\begin{array}{c}
f \ (\text{Fix} \ f)
\downarrow \\
\map{\text{fold} \ g}
\quad
\map{\text{fold} \ g}
\end{array}
\quad
\begin{array}{c}
f a
\downarrow \\
\text{g}
\downarrow \\
\text{fa}
\end{array}
\]

Using this diagram and that fact that $\text{In}$ is the inverse to out, the fold function itself can be defined as follows:

$\text{fold} \ g = g \ . \ \text{map} \ (\text{fold} \ g) \ . \ \text{out}$

That is, the function $\text{fold} \ g$ first applies the function out to break down an argument of the recursive type $\text{Fix} \ f$ into a structured value of type $f$ ($\text{Fix} \ f$), then applies the function $\text{map}$ ($\text{fold} \ g$) to recursively process each of the $\text{Fix} \ f$ components to give a value of type $f \ a$, and finally applies the algebra $g$ combine all the $a$ components into a single result value of type $a$. In this manner, fold is a general purpose function for processing values of a recursive type using a simple pattern of recursion.

While previously we defined fold functions that were specific to particular recursive data types (for example, expressions), the above version of fold is polytypic. Moreover, the definition for fold is precisely dual to that for unfold. Hence, taking an abstract approach to recursion operators has revealed an explicit duality between fold and unfold that might otherwise have been missed.

Returning to processes, because the datatype $\text{Proc}$ is expressed as the least fixpoint of the functor $P$, and the combining function is expressed as an algebra $\text{comb}$ for $P$, the denotational semantics of processes as trees can now be defined using the polytypic version of fold:

$\text{eval} :: \text{Proc} \to \text{Tree}$
$\text{eval} = \text{fold} \ \text{comb}$

In summary, we have now expressed the denotational semantics of processes as trees as the unique strict function $\text{fold} \ \text{comb}$ that makes the upper square in the diagram below commute, and dually, the operational semantics of processes as trees as the unique function $\text{unfold} \ \text{trans}'$ that makes the lower square commute.

8 Reasoning about CCS

We have now defined both operational and denotational semantics for CCS. It is natural to ask how the two semantics are related. In this section we show that they are equal by exploiting the universality of the recursion operator $\text{fold}$.

8.1 Universality of fold

For arbitrary recursive data types expressed as the least fixpoint of functors, the universal property of $\text{fold}$ is captured by the following equivalence (for strict $h$): [13, 14]:

$g \ . \ \text{map} \ h = h \ . \ \text{In} \ \Leftrightarrow \ h = \text{fold} \ g$

Because $\text{fold}$ is a polytypic operator, so this universal property is a polytypic proof principle [3]. Returning to our semantics for processes, it is easy to verify that $\text{unfold} \ \text{trans}'$ is strict, using the definitions of the functions concerned, together with the strictness of tags defined using $\text{newtype}$. Hence, applying the universal property gives:

$\text{unfold} \ \text{trans}' = \text{fold} \ \text{comb}$

$\quad \text{eval} \ . \ \text{map} \ (\text{unfold} \ \text{trans}') = \text{unfold} \ \text{trans}'$. In the final equation above can now be verified by a routine induction on the size of an argument $p :: P \ \text{Proc}$, where the size is defined as the number of $\text{In}$ tags in $p$. Hence our operational and denotational semantics for CCS are equal for all processes of finite size. Further work is still required to extend the proof to processes of infinite size.

The two semantics can also be proved equal using the dual universal property of $\text{unfold}$ rather than that of $\text{fold}$, but the proof works out simpler using the later.

9 Summary and future work

In this paper, we have shown how fold and unfold can be used to structure and reason about program semantics within Haskell. The paper is based upon categorical work on semantics, but explaining the ideas using Haskell makes them simpler, accessible to a wider audience, and executable. Interesting topics for future work include:

- The application of the techniques to further examples, including languages in different paradigms;
- The use of monadic fold operators to structure the denotational semantics of programming languages with imperative effects such as mutable state;
- Exploring recursion operators and algebraic properties that correspond to non-structural patterns of induction, such as induction on the size of values;
- Further applications of fold and unfold.

Acknowledgements

This work is supported by Engineering and Physical Sciences Research Council (EPSRC) research grant GR/L74491 Structured Recursive Programming. Thanks to colleagues in Birmingham, Cambridge, Glasgow, Nottingham, Oxford, and York for many useful comments and suggestions.
References


