Coalgebraic Techniques for Reactive Functional Programming

Richard B. Kieburtz

Oregon Graduate Institute,
20000 N.W. Walker Road,
Beaverton, OR 97006, USA
dick@cse.ogi.edu

Abstract

A reactive process can be abstractly modeled as a finite transition system in which sets of reactions are characterized by states. Such systems are naturally modeled by coalgebras. They admit reasoning by coinduction and related techniques. This paper introduces coalgebraic modeling of reactive programs and gives examples of its use in reactive functional programming.

1. Reactive Systems

The usual academic introduction to computing emphasizes the calculation of results from initial data. The student is taught to use recursive programming techniques and to worry about termination of programs.

Many, if not most computer applications are quite different. Their programs are not expected to terminate, ever. Data for these programs arrive almost continually, without beginning or end. These aspects are typical of embedded software controllers for physical systems, communications switches, and computer operating systems. Nor do such programs calculate results, in the normal sense of the word. They deliver control signals to be consumed by their surrounding environment. Programming such systems successfully requires, at the least, a different perspective than does calculation.

When we refer to a system, we have in mind a networked collection of processes that interact through streams of events. This general and somewhat imprecise notion allows processes to be realized by computer programs in execution, by hardware devices, data sensors, humans or subsystems that may be comprised of all of these forms. We're going to study some techniques for programming and reasoning about programmed processes.

A *reactive* process is one whose behaviors evolve incrementally, through a series of responses to discrete *events* manifested in its environment. Each response (or reaction) to an event is determined by the specific nature of the event and by the state of the process. A response may consist of a change of state, or may launch a new event into the environment, or may do both.

A process is said to behave *synchronously* if both

1. the response of the process depends only upon
    - the event it receives from its environment and

- the process state when the event is received;

2. the response of the process to an event is always deposited in the environment before the set of events in the environment is changed by external influences.

Th importance of synchronous processes has been emphasized by Gerard Berry [Ber89, BB91].

For all practical purposes, a synchronous reactive system may be considered to compute its responses instantaneously, as no experiment formulated by an observer of events in the environment can determine the actual times taken by a synchronous system to respond. If the experimenter were able to devise alternate timings for the occurrences of events in a a given sequence, and the altered timings caused the system to produce different responses, this would be evidence that the system is not actually synchronous in its behavior.

The synchrony hypothesis does not hold of all real systems, but it does hold for many computer-controlled systems because the latency of external devices or subsystems is longer than the time needed for a controller to compute responses.

The synchrony hypothesis is of particular importance because it allows programming of each process under an assumption that the state of its environment is stable during a computation step of the process. If the synchrony hypothesis cannot be justified, processes must be governed by protocols that allow for uncertainties in the state of the environment or their behavior may be unpredictable. Such programming is intellectually far more demanding. Synchronous reactive programs can be constructed by following principles that are common to to all iterative programming. These principles support formal reasoning about programs as they are designed. We intend to introduce you to some of these principles here.

Since most reactive systems have internal state, it may seem curious to talk about functional programming as a paradigm for building a reactive control system. However, the state changes of a synchronous reactive system are functionally related to the event that triggers a reaction and to the previous value of the state. Expressed equationally, $S' = F(e, S)$, where S represents the value of the state at the occurrence of an event, e; S' indicates the value of the state the next observation after the event and F is a function, called the *transition function.* Thus functional programming techniques can be used. Only the interaction of a controller with its environment requires additional explanation.

Conceptually, we can think of a reactive system as a set of processes that communicate through an environment in which processes may consume and deposit tokens that we shall call *events.* Without being too specific about the exact nature of an event, we shall use this term to designate a signal that can be detected by a process. We also assume that events are tagged, so that a process is able to discriminate among a finite set of possible events.

The environment may consist of channels, each linking a specific set processes or it may just be a common pool of events visible to all processes. Each process can discriminate, via its input ports, among a finite set of input events and may deposit, at its output ports, one or more output events. In addition, we shall assume that these processes each have internal state, i.e. the behavior of a process in response to an input event may be history-sensitive. However, we assume that the state of each process is entirely hidden from other processes except for such information as the process communicates by its behavior; that is, by the output events it deposits in reaction to the input events with which it is presented. It is convenient to model these processes as transition systems.

1.1. State-based transition systems

A reactive process can be abstractly modeled as a state-based transition system, $\langle S, E, \sigma \rangle$, where

S is set of state values (not necessarily finite),

E is a finite set of tokens, representing events that may occur in the environment, and

$\sigma : S \times \mathcal{P}(E) \rightarrow S \times \mathcal{P}(E)$ is a transition relation.

A transition system is said to be deterministic if its transition relation is a (partial) function, otherwise it is nondeterministic. Although most of the systems we shall wish to specify will be deterministic, it is useful to allow for nondeterministic systems. If a particular source of input events to a process were invisible to an external observer, for instance, that process might appear to behave nondeterministically.

A program for a reactive process must specify three things:

- the type of its state values, S;
- the set of input and output events that the process recognizes;
- its transition relation.

Clearly the most demanding task of programming a process is to give a finite presentation of the transition relation. A simple form for this presentation is a finite set of guarded transition elements. To represent the state, we assume that there is a state variable, X. Although the state variable might have any type, it is often useful to think of it in terms of a record type. A guarded transition element has the form

$$\langle \mathit{trigger} \rangle \ : \ G \Rightarrow (X',\ \mathit{out}) := F(X)$$

where

trigger is an enumeration of a set of events drawn from E,

G, called the guard, is a conjunction of atomic equalities and disequalities over the state variable X,

$F : S \rightarrow S \times \mathcal{P}(E)$ is a typed expression representing a partial transition function, and

out is a distinguished variable name that represents a set of output events that result from the transition.

A transition element is said to be *enabled* in an environment, $\mathcal{E}$, if $\mathcal{E}$ contains exactly the set of events specified in the trigger of the transition element and its guard evaluates to `true` when the current value of the state is substituted for the state variable, X. An enabled transition element can fire. When a transition element fires, the process consumes the set of trigger events from its evironment, produces the state transition given by the assignment of $\mathit{fst}(F(X))$ to the state variable and deposits in its environment the set of output events given by $\mathit{snd}(F(X))$. These three activities are assumed to occur simultaneously.

Note that if the trigger set of a guarded transition element is empty, then the transition is enabled solely by the guarded condition on the process's state. Such a transition is said to be *spontanteous.* A spontaneous transition element that generates no output event may fire repeatedly, for as long as its state transition continues to satisfy its guard. (If the process were to produce an output event, then the trigger condition required for its repeated firing would no longer be satisfied.) Thus a reactive process with a spontaneous transition is capable of purely iterative computation, potentially without bound. An unbounded iterative computation by a deterministic process corresponds to the death of the process, as it can never participate in another interaction with its environment.

To complete the specification of a transition relation, we may sometimes want to constrain the initial valuations of the state. This can be specified with a guard expression prefaced by the keyword *init.*

A set of guarded transition elements specifies a deterministic transition system if at most one transition element can be enabled for any valuation of the state. This condition is stronger than necessary, because multiple elements might fire, producing the same state transition and output events, or because not all valuations of the state may be reachable from a constrained set of initial valuations.

1.2. Iterative computations

We have seen how a spontaneous transition element can lead to "silent" iterative computation by a process. Event-driven transition elements can also participate in iterative computations, although they cannot be silent. The output events deposited by the firing of one transition element may produce a set of events that enables firing of another element. A process that continues to generate event configurations that sustain firing of its own transition elements will perform an iterative computation. Clearly, a set of several communicating processes can also perform iterative computation as they exchange enabling event configurations.

Control of iterative computation is effected by programming the guards and transition functions of a set of guarded transition elements. It is this mode of program specification that we intend to dicuss and to reason about. From a logical point of view, the important properties are the *invariant* conditions that hold of the state during an iteration, and the *termination* conditions on the state that disable an iteration. Characterizing the invariants and termination conditions is the key to understanding iterative computation. In the following sections, we shall study some specific iteration structures and the forms of their invariants.

1.3. Example: A reset counter

To illustrate the specification of a transition system we shall consider a very simple example. A reset counter accepts a stream of input events, each of which has either the tag *Zero* or *One*. When it receives a *One*, the process increments an internal counter; when it receives a *Zero*, it resets the counter's value to zero and if its previous value was non-zero, outputs that value. The counter can be specified as a transition system as follows:

$$
\begin{array}{l}
\textbf{input } \mathit{Zero}, \mathit{One};\\
\textbf{output } \mathit{Count} \textbf{ of } \mathit{int};\\
\textbf{var } X : \mathit{int};\\
\textbf{init } X = 0;\\
\textbf{process}\\
\quad\ \{\mathit{Zero}\} : X = 0 \Rightarrow X' = X\\
\mid\ \ \{\mathit{Zero}\} : X > 0 \Rightarrow X' = 0, \textbf{emit } \mathit{Count}(X)\\
\mid\ \ \{\mathit{One}\} : \mathit{true} \Rightarrow X' = 1 + X
\end{array}
$$

In this specification, the first two transition elements are triggered by exactly the same event, but non-overlapping guard expressions determine different responses to this event. These guards test the state of the process. Since there are only finintely many guards and since the guards cannot overlap on any common input event, the guards themselves can be used to characterize a finite set of states of the system, in this case, two states.

If we were to partition the transition elements by these finite states, the guards could be eliminated. Some transition elements may need to be duplicated in the partition, as their guards span more than one state. In the example, this is the case with the transition on the event *One*, whose guard is *true*. Of course, when the partition is made, it will be necessary to explicitly designate the target state of each transition, i.e. the state in which the post-transition value of the state variable will satisfy the guard.

For the reset counter, we partition the transition elements into two sets (states), which we shall call *reset* and *counting*. The transition elements, rewritten without explicit guards, become

$$
\begin{aligned}
&\mathit{reset}\ \{ \quad \{\mathit{Zero}\} \Rightarrow X' = X,\ \textbf{state}{=}\mathit{reset}\\
&\qquad\quad\ \mid\ \{\mathit{One}\} \Rightarrow X' = 1 + X,\ \textbf{state}{=}\mathit{counting}\}\\[1ex]
&\mathit{counting}\ \{ \quad \{\mathit{Zero}\} \Rightarrow X' = 0,\ \textbf{emit}\ \mathit{Count}(X),\ \textbf{state}{=}\mathit{reset}\\
&\qquad\qquad\ \mid\ \{\mathit{One}\} \Rightarrow X' = 1 + X,\ \textbf{state}{=}\mathit{counting}\}
\end{aligned}
$$

In this form, it is apparent that the reset counter process is purely event-driven.

1.4. Example: A sliding-window protocol

A sliding-window protocol is a two-party communications protocol that supports transmission of a series of message packets in a prescribed order, tolerates transmission delays and is robust under occasional packet loss. To support these properties, message packets are buffered at the sending end until their receipt has been confirmed by an acknowledgement sent back by the recipient. Only a bounded number of buffer slots is provided, however. We shall call the two communicating processes the *Sender* and the *Receiver*. We shall give a specification for the *Sender* as a state transition system.

A message packet comprises a header, a message body, and a parity-check sum. The header must contain a sequence number that uniquely identifies the message packet. The header can also contain other information. In our specification, it contains the least sequence number of a packet whose receipt has not been acknowledged, or more precisely, whose acknowledgement has not been received by the *Sender*. Acknowledgement packets are sent back from the *Receiver* to the *Sender*. An acknowledgement packet consists of a header, containing the largest sequence number such that every packet with that number or a lesser number has been received, and a parity check sum.

If ever a corrupted packet is detected, i.e. one for which the parity check fails, it is simply discarded. Also, a packet that arrives out of order or with a sequence number that duplicates one already received is discarded. If a sent packet has not been acknowledged for a time interval that exceeds a prescribed limit, the packet is presumed to have been lost and is resent. Similarly, an acknowledgement may be resent. The protocol determines when retransmission of a message packet or an acknowledgement is necessary, and when a buffer slot can safely be released by the *Sender*.

Since the number of buffer slots is bounded, there is a limit to how many messages can be sent without being acknowledged. When no new message packets can be accepted, the channel is logically blocked until receipt of some of the extant packets has been acknowledged. Let N be the sequence number assigned to the most recent packet the sender has accepted from its driver. Let L be the highest sequence number that has been acknowledged. Let B be the number of buffer slots available. Clearly, the inequality $B \geq N - L$ constrains acceptance of new message packets by the *Sender* process. The *Sender* maintains the values of N and L in fields of a state variable.

To obtain a sense of time, the *Sender* also reacts to a clock tick event, *Tick*. The *Sender* maintains a count of clock ticks in the *Time* field of its state variable.

We assume that messages are furnished to the *Sender* process by a driver, which deposits a message-valued event, *NewMessage*, in the *Sender*'s environment to request the transmission of a new message packet. The *Sender* acknowledges the driver by depositing either an *Accept* event or a *Reject* event into the environment. In accepting a message body from its driver, the *Sender* makes an implicit guarantee that the message body will eventually be delivered to the *Receiver*, in the order in which it was accepted. If the *Sender* rejects a message body submitted by its driver, it guarantees that it will not deliver it.

$$
\begin{array}{l}
\textbf{input } \textit{NewMessage} \textbf{ of } \textit{Msgtype}, \textit{Ack} \textbf{ of } \textit{int}, \textit{Tick}; \\
\textbf{output } \textit{Msg} \textbf{ of } \{\textit{LastAck} : \textit{int}; \textit{SeqNumber} : \textit{int}; \textit{Body} : \textit{Msgtype}\}, \\
\qquad \textit{Accept}, \textit{Reject}; \\
\textbf{var } X : \{L, N, \textit{Time} : \textit{int}; \textit{Buffer} : \textbf{array}[B] \text{ of } \textit{Msgtype}\}; \\
\textbf{init } X.L = 0,\ X.N = 0,\ X.\textit{Time} = 0; \\
\textbf{process} \\
\quad \{\textit{NewMessage}(S)\} : (X.N - X.L) = B \Rightarrow X' = X, \textbf{ emit } \textit{Reject} \\
\mid \quad \{\textit{NewMessage}(S)\} : (X.N - X.L) < B \Rightarrow \\
\qquad\qquad \textbf{let } n = 1 + X.N \textbf{ in} \\
\qquad\qquad X' = X \oplus \{N = n;\ \textit{Time} = 0; \\
\qquad\qquad\qquad \textit{Buffer}[n \bmod B] = S\}, \\
\qquad\qquad \textbf{emit } \textit{Msg}\{\textit{LastAck} = X.L; \\
\qquad\qquad\qquad \textit{SeqNumber} = n, \\
\qquad\qquad\qquad \textit{Body} = S\}, \\
\qquad\qquad \textbf{emit } \textit{Accept} \\
\mid \quad \{\textit{Tick}\} : X.\textit{Time} < \textit{MaxTime} \Rightarrow X' = X \oplus \{\textit{Time} = 1 + X.\textit{Time}\} \\
\mid \quad \{\textit{Tick}\} : X.\textit{Time} \geq \textit{MaxTime} \wedge (X.N - X.L) > 0 \Rightarrow \\
\qquad\qquad X' = X \oplus \{\textit{Time} = \textit{MaxTime} * (B-1)/B\}, \\
\qquad\qquad \textbf{emit } \textit{Msg}\{\textit{LastAck} = X.L; \\
\qquad\qquad\qquad \textit{SeqNumber} = 1 + X.L; \\
\qquad\qquad\qquad \textit{Body} = \textit{Buffer}[(1 + X.L) \bmod B]\} \\
\mid \quad \{\textit{Tick}\} : X.\textit{Time} \geq \textit{MaxTime} \wedge (X.N - X.L) = 0 \Rightarrow \\
\qquad\qquad X' = X \oplus \{\textit{Time} = 0\} \\
\mid \quad \{\textit{Ack}(M)\} : M > X.L \Rightarrow X' = X \oplus \{L = M;\ \textit{Time} = 0\} \\
\mid \quad \{\textit{Ack}(M)\} : M \leq X.L \Rightarrow X' = X
\end{array}
$$

The notation $X \oplus \{\cdots\}$ designates a new value of the state whose fields retain the values given by the state variable, X, except for the field values explicitly specified between the braces.

The first transition element triggered by the *NewMessage* event is the case in which the *Sender*'s buffer is full of unacknowledged packets; no more can be accepted. The second element handles the case in which the buffer is not full; a new message body is accepted, given the next sequence number, and sent in a packet. A copy of the message body is buffered awaiting acknowledgement of successful receipt of the packet.

The first transition element triggered by the *Tick* event covers the cases in which the accounted time has not reached the limit, *MaxTime*. It specifies that the time counter field of the state variable is incremented. The next two transition elements cover the case that the time allotted to receive an acknowledgement of a previously sent packet, or since the last acknowledgement was received, has expired. This time measurement is introduced in the protocol to avoid a deadlock situation in which both *Sender* and *Receiver* might be indefinitely awaiting an event to be deposited by the other. Upon the expiration of accounted time, the *Sender* will resend a message packet if any remains unacknowledged.

The transition elements triggered by the *Ack* event update the last acknowledged sequence number in the *Sender*'s state, effectively releasing one or more buffer slots. An effective aknowledgement also resets the *Time* count in the *Sender*'s state.

1.5. Functional Reactive Programming

Reactive programming is neither intrinsically functional nor imperative. Although the more traditional approach to programming reactions has been to use state mutation in an imperative style, this has always required care to avoid interfering effects when concurrent processes may be involved. An ideal reaction must be an atomic action.

In a functional realization, a reaction is an effects-free response to the set of input events in a program's environment when it is activated. An output event may have non-invertible consequences once it has been emitted, but the reaction which emits it does not. The program emits an event that may be interpreted as a command in the environment—the program does not carry out the interpretation. Among early reactive programming languages, *Esterel* takes an imperative programming approach while *Lustre* and *Signal* are based upon a more nearly functional approach. However, none of these reactive programming languages is a wide-spectrum programming language.

There is no reason not to employ a powerful, functional programming language in reactive programming. We can realize the advantages of elegance and ease of reasoning associated with functional programming style, while applying it to control time-dependent, active processes. One of the first to develop a functional, reactive programming style was Conal Elliott [EH97] who used Haskell to script a library of graphics animation primitives. More recently, functional reactive programming has been applied to robot control [HPE99]. A reactive programming approach is described in a new Haskell-based programming textbook by Paul Hudak [Hud99].

2. Streams of Events

The notion of a *stream* is fundamental in programming and reasoning about discrete, deterministic systems. A stream is a potentially infinite sequence (of events, values, observations, etc.) whose elements are ordered in time. This does not imply that the elements of a stream are actually timed, i.e. they don't need to occur at particular times, it merely implies a total ordering of the elements that is compatible with temporal order. An ordering may be implied by a deterministic rule for computation of a next element from the ones that have preceded it. If we assume that this computation takes time, then this ordering of computed elements is compatible with temporal order.

A stream has two access methods: one to access the current element, or stream head, and another to access the remainder of the stream, called the stream tail. By repeatedly applying the stream tail operator, followed by an access of the stream head, one can reach any finite element of a stream.

Here are some examples of streams:

- the ticks of a clock (with a discrete mechanism),
- the natural numbers, ordered by the computation of a successor,
- the Fibonacci sequence,
- a sequence of *NewMessage*, *Tick*, or *Ack* events that trigger responses of the *Sender* process in the sliding-window protocol discussed in Section 1.4,
- a sequence of values assumed by the state variable, X, of the *Sender* process in the sliding-window protocol under transitions induced by a stream of input events.

If you are familiar with functional programming languages that have lazy evaluation rules, such as Haskell or Miranda, you may associate streams with infinite lists. An infinite list is a good example of a stream, as its elements have a specified type, it is unbounded in extent, and its elements are accessible only in sequence.

However, a stream does not have to be represented as a data structure. In fact, an infinite list isn't really a proper data structure; it cannot be fully elaborated with finite computing resources.

A fundamental difference between lists and streams is that a list is characterized by its *construction* and a stream by its *observation.* A list is constructed either as the (unique) empty list or as the pairing of an element value with an already constructed list. Observations of a list must be defined in terms of its construction. Dually, a stream is observed through its two access methods; its construction is specified only insofar as a representation is needed for effective definition of the access methods. A lazily constructed list blurs this distinction by using a representation that defers construction of the list until it is demanded by a request for an observation.

As an example, here's a finite representation of the stream of natural numbers:

$$Nats = ((\lambda x.1 + x), 0)$$

In this representation, the stream head operator is *snd*, which takes the second projection of a pair. The stream tail operator is a bit more complicated, it is $\langle fst, ap\rangle$, where *ap* is the function application operator, defined by the equation $ap\,(f, x) = f\ x$. The stream tail operator applied to *Nats* yields the stream represented by $((\lambda x.1 + x), 1)$. The stream head and stream tail operations are total functions, defined on every instance of this stream representation. Notice that in this example, the stream representation doesn't resemble a list.

2.1. A logic for streams

In a first-order predicate logic, the object variables are given interpretations over "small" sets. This affords great generality of models for satisfiable logical formulae. The universal quantifier implicitly quantifies over the universe of such sets. Streams can be modeled as enumerable sets with a total ordering, so they too can be described in terms of ordinary predicate logic. However, the ordering must be axiomatized in the logic in order for a formula to describe only streams, and this is not always convenient.

It would be convenient to have a logic for reasoning about streams, in which the total ordering of elements was implicit. Such logics have been extensively investigated [MP91]. A widely accepted one is called *Linear Temporal Logic*, abbreviated as LTL. LTL is a *modal* logic, in which the modality restricts models of its formulae to streams. It is a syntactic system, however, and while it helps intuition to think about streams as its models, the structural rules of the logic are complete on their own.

LTL syntax includes the boolean connectives $\neg$, $\wedge$ and $\vee$, a countable set of object variables, and countable sets of N-place predicate symbols for arities $N = 1, 2, 3, \cdots$.

LTL has a set of modal operators not found in universal first-order logic. These operators preface formulae that can be interpreted over ordinary sets, changing their interpretations to characterize properties of stream elements. When an ordinary formula is prefaced by one of the temporal operators, it characterizes a property of a stream of elements. The temporal operators form temporal formulas from ordinary formulas. These operators are:

$\Box$: pronounced "always"; $\Box P$ asserts that the property P holds over all elements of a stream;

$\mathbf{X}$: pronounced "next"; $\mathbf{X}P$ asserts that the element immediately following the stream head has property P.

$\mathbf{U}$: pronounced "until"; this operator separates two formulae. $P\,\mathbf{U}\,Q$ asserts that property P holds of all elements in the initial segment of a stream up to the first element which satisfies property Q.

An ordinary formula can also be interpreted as characterizing a property of a stream of elements. A formula P, interpreted as a temporal formula, holds of a stream if the initial element of the stream satisfies P.

Commonly, LTL formulas make use of an additional temporal operator that is derived from the $\Box$ operator with the use of logical negation:

$\Diamond$: pronounced “eventually”; $\Diamond P$ asserts that a sequence contains a finitely reachable element for which the property P holds. The operator is defined by $\Diamond P \equiv \neg\Box(\neg P)$.

The immediate utility of LTL formulas is that they provide us with a language for expressing logical properties of streams.

3. Induction and Coinduction

Functional programmers are accustomed to reasoning about instances of finite, but unbounded, data types that are defined by recursion. Recursive equations provide a bounded specification for data objects whose size is unbounded. Let U be some universal set, and let $F : \mathcal{P}(U) \to \mathcal{P}(U)$. A recursion equation $X = F\,X$ has a solution over $\mathcal{P}(U)$ (called a *fixed point* of F) if F is monotonic, i.e. if $F\,X \subseteq F\,Y$ whenever $X \subseteq Y$. In fact, it may have more than one fixed point.

You may be familiar with fixed points from studying denotational semantics. That theory introduces partially ordered sets as domains in which to interpret recursively defined programs. Here, we look at a far simpler domain: “small” sets with no assumed structure. However, we shall not concern ourselves about the possibility of partial functions due to non-terminating function definitions.

The following definitions are taken from [Gor95].

Definition 3.1: A set $X \subseteq \mathcal{P}(U)$ is *F-closed* if $F\,X \subseteq X$

Definition 3.2: A set $X \subseteq \mathcal{P}(U)$ is *F-dense* if $X \subseteq F\,X$

Definition 3.3: Least and greatest fixed points.

$$\mu X.F\,X =_{\text{def}} \bigcap\{X \mid F\,X \subseteq X\}$$

$$\nu X.F\,X =_{\text{def}} \bigcup\{X \mid X \subseteq F\,X\}$$

□

The least fixed point of F, abbreviated as μ_F, is the smallest *F-closed* set and the greatest fixed point, ν_F, is the largest *F-dense* set in $\mathcal{P}(U)$.

Example: Let $F : \mathcal{P}(U) \to \mathcal{P}(U)$ be the function defined by $F\,X = \{\mathit{Succ}(x) \mid x \in X\}$. Then *Nat* is an F-closed set. In fact, it is the least F-closed set, up to isomorphism. The isomorphism is based upon the observation that all singleton sets are isomorphic to one another, and thus to the set $\{\mathit{Zero}\}$.

3.1. Induction

The principle of induction, and its dual, the principle of coinduction allow us to reason, with finite formulas, about properties of least and greatest fixed points. Let's start with induction, which is more familiar. Recognizing that a set can be defined by a characteristic predicate, let P be a one-place predicate.

If P is F-closed, then the least fixed-point of F, μ_F, is a subset of P; thus $\forall x \in \mu_F.\,P(x)$. This is known as the principle of induction. Letting F designate the pointwise transformation of members of a set, we can express the induction rule in the form it is most often seen, as a logical sequent:

$$\frac{P(x) \Rightarrow P(F\,x)}{\forall x \in \mu_F.\,P(x)}$$

or it can be written as an implication of subset relations,

$$(F\,P \subseteq P) \Rightarrow (\mu_F \subseteq P)$$

Example 3.1.1: List-induction. A list datatype is defined by its construction. To define a list data type in ML-like syntax (except that type constructors are postfix operators in ML) we could write

$$\textbf{datatype}\ \mathit{List}(A) = \mathit{Nil}\ \mid\ \mathit{Cons}\ \textbf{of}\ A\ *\ \mathit{List}(A);$$

In set notation, this declaration corresponds to a recursion equation of the form $\mathit{List}(A) = F\,\mathit{List}(A)$, where $F\,X = \mathbf{1} + (A \times X)$ and $\mathbf{1}$ denotes a singleton set. The least fixed-point of this equation is the set of (finite) lists of type-A elements constructed with *Nil* and *Cons*.

A rule for list induction follows directly from the datatype declaration. There are two hypotheses, one for each of the list constructors.

$$\frac{P(\mathit{Nil}) \qquad (a \in A) \Rightarrow P(x) \Rightarrow P(\mathit{Cons}(a, x))}{\forall x \in \mathit{List}(A).\, P(x)}$$

□

3.2. Coinduction

The coinduction principle is dual to the principle of induction; formally, the set inclusions in its set-based formulation are reversed,

$$(P \subseteq F\,P) \Rightarrow (P \subseteq \nu_F)$$

expressing that if P is a predicate characterizing an F-dense set, then it is a subset of the greatest F-dense set, ν_F.

A logically equivalent formulation as a sequent expression is:

$$\frac{P(F\,x) \Rightarrow P(x)}{\forall x.\, P(x) \Rightarrow x \in \nu_F}$$

The coinduction principle doesn't have quite as obvious a programming interpretation as does the induction principle, however. The implicand does not say that that every element of ν_F has the property P, but rather that ν_F is a *cover* for the property P. It is not useful to think of a greatest F-dense set as a datatype, as it is not characterized in terms of a construction of data elements. Rather, we may call it a type of *codata*, characterized in terms of the methods by which it can be observed.

In imperative programming, we compose methods of observation to define a computation. The reactions of an event-driven system are instances of methods of observation of the system. codata are as fundamental to imperative programming as data are in a purely recursive programming style.

Example 3.2.1: Stream coinduction.
By analogy to the definition of a *List* datatype in ML, we write in ML-like notation (you can't actually declare codata in ML) the declaration

$$\textbf{codata}\ \mathit{Stream}(A) = \{\mathit{Shd} : A,\ \mathit{Stl} : \mathit{Stream}(A)\};$$

The identifiers appearing between the curly braces are those of the *witness* functions of the codata type. Each witness function has $\mathit{Stream}(A)$ as its domain type and the type declared after the colon as its codomain. The declaration corresponds in set notation to a recursion equation $\mathit{Stream}(A) = F\,(\mathit{Stream}(A))$, where $F\,X = A \times X$. This recursion equation has a greatest fixed-point solution in sets, as there are F-dense sets for this particular F. The set of infinite sequences of A-typed values is

such a set.
□

Instances of stream-typed codata are familiar in programming. The most ubiquitous example is that of a sequential input file. The operation that reads a current value from the file buffer corresponds to the witness function *Shd* applied to a stream; the operation that gets the next value in sequence into the file buffer corresponds to *Stl* applied to a stream. A sequence of externally generated events that drives a reactive program is another instance of stream codata. In both these examples, the program does not specify how the stream is generated; it specifies an observation, or reaction to each of the stream's elements.

The coinduction principle formulated in terms of properties of codata, as given above, provides implicative forms for characterizing a stream, but these forms are less useful for reasoning about the observations of a stream. Suppose we consider predicates over the *observations* (or contexts, or continuations) of streams, rather than predicates over streams themselves. In the formulation below, the free variable k represents a hypothetical observation, or program continuation. In a continuation-based, or iterative formulation of a program, the effect of a program segment, r, is to extend the hypothesized continuation to the new continuation

The stream coinduction principle becomes:

$$\frac{R(k) \Rightarrow R(k \circ \mathit{Stl})}{\forall i.\, R(k \circ \mathit{Stl}^i\,)}$$

where Stl^i denotes the i-fold iteration of the stream tail operator. In this formulation, the predicate R is an iterative invariant of the stream. The antecedent in the formula is an assertion that the property predicated by R is invariant under an extension of the observation to the tail of a stream. The consequent asserts that the property holds for an extension of the observation to arbitrarily many stream tail operations.

An equivalent way to state the stream coinduction principle is in terms of the LTL operator, □ (pronounced "always").

$$\frac{R(k) \Rightarrow R(k \circ \mathit{Stl})}{\Box\, R(k)}$$

Of course, when stream coinduction is applied to a particular instance of stream codata, the *Stl* operator is substituted by a term that represents the programmed witness of the tail for that particular instance. The continuation variable k can also be substituted by a term representing a programmed reaction to an event, or stream element. If you are familiar with the weakest-precondition logic of Dijkstra [Dij76], you may see a similarity between stream coinduction in its continuation-based formulation and the verification condition of an iterative program, such as a **for**-loop, in weakest-precondition logic.
□

4. Iterative programming

In a purely iterative programming style, the programmer's attention is focused on control, rather than data. Instead of constructing values, an iterative program consumes them. Instead of calculating intermediate results with recursion, an iterative program dynamically extends the execution of a program by repetition. Instead of tagging alternate forms of a result to distinguish them, an interative program selects among alternate program segments to consume an intermediate value.

Iterative calculation is based upon the mathematical notion of a program *continuation.* A continuation is a function that takes a current value, say of type A, and consumes it to produce an ultimate result. We can represent an A-consuming continuation by a variable typed as $k : A \to R$, where R is the type of results, which may or may not be specified. For instance, an example of an *int*-consuming continuation would be $print\_int{:}int \to R$, an operation that consumes an integer value, converts it into a character string representation, and writes it to a file or prints its image on a screen. The type of the final result produced by *print_int* is not meaningful and is left unspecified.

As an intellectual exercise, let us briefly consider programming with explicit continuations, although this is almost never done in real programming. When programming with explicit continuations, a variable ranging over continuations is often used to represent the context of a calculation. A programmed function extends the continuation, yielding a context for parameters of the function. Given an ordinary functional program, written as

$$\textbf{let}\ f\ x = \langle expr\rangle;$$

its iterative equivalent is given an extra continuation argument to represent the context and written as

$$\textbf{let}\ f'\ x\ k = \langle k\ expr\rangle;$$

Notice the difference. An application of the function f' is not interpreted as returning a value in its textual context; an evaluation of the body, $\langle expr\rangle$, is directly consumed by the continuation, k.

Suppose the original function is typed as $f : int \to int$. Then a program to print the result of the application $f\ 5$ is $print\_int\,(f\ 5)$. Upon substitution of the argument 5 for the bound variable in the body of the definition of f, this reduces to $print\_int\,\langle expr\rangle[5/x]$.

In the iterative style, we would write an equivalent program as $f'\ 5\ print\_int$. This program reduces to the same calculation as does the functional program, after substituting two arguments into the definition of f'. The iterative style program differs from the conventional functional style in that programmed expressions are parameterised with respect to continuations, allowing for the possibility that a continuation other than the "normal" one derived from the textual context of the expression might be used.

Suppose, in an application of f', that only the intended continuation were given, and the value argument was not. Upon substituting the continuation argument into the body of f', we obtain a new continuation, $\lambda x.\, print\_int\langle expr\rangle$. The programmed definition of f' has been used to extend the continuation *print_int* to a continuation that first applies f to the value it consumes, then sends the result to *print_int*. This illustrates a fundamental aspect of iterative programming style.

In purely functional programming, no benefit is gained by making the program continuation explicit. The "normal" continuation for an expression is that provided by the textual context of the expression. Programming with continuations becomes useful only when alternate continuations are needed, as for instance, to account for exceptions or for backtracking control or to extend functional programming techniques to applications such as state transition systems, for which iterative programming is a more natural style.

When programming a finite-state transition system, a program segment is created for each state by programming a reaction of the system to each of the input events visible in that state. A reaction extends one or more of the continuations that might consume data from the particular state. The reaction may produce an input event for another state of the same system or an output event for consumption by the external environment. Without the concept of continuations to program reactions involving state transitions, it would be difficult to program a finite transition system without the artifice of coding control states as data.

4.1. Exceptions and continuations

Exceptions as a programming concept represent alternative continuations. Although programming language definitions are notoriously inconsistent about how exception names are bound to continuations, such a binding always exists. Even an undefined exception, if raised, is bound by default to a continuation whose result is to terminate the program and (sometimes) display an error message.

In the simplest case, an exception name may be statically bound to a programmed continuation with a handler expression. If the exception name is ever raised by executing a **raise** command for the named exception, within the scope of a handler, then the alternate exception defined for that exception name by the handler is invoked.

When continuations are made explicit in programs, a continuation can be defined and bound to a continuation-typed identifier. When that identifier is passed to a program component, the program component gains access to the continuation. Explicitly invoking an alternate continuation in a calculation is semantically equivalent to having raised an exception whose name was bound to that continuation.

4.1.1. Continuation-passing style

The transformation of expressions from a lambda-calculus into a continuation calculus is well known as the *CPS transformation.* CPS is an abreviation for "continuation-passing style". In the table below, we give the CPS transformation for a lambda calculus extended with atomic constants, pairs and exceptions. The exception constructor is *Exc.* Primes designate expressions in CPS form that manifests continuations. Because exceptions introduce optional continuations, each continuation expression is here represented as a pair whose first component is the normal continuation and whose second component is the exceptional continuation.

$$\begin{aligned}
(\lambda x.\, e)' &= \lambda k.\, e'\; k \\
(f\; e)' &= \lambda k.\, e'\; (\lambda v.\, f'\; v\; k) \\
c' &= \lambda k.\, \mathit{fst}\, k\; c \\
(e_1,\, e2)' &= \lambda k.\, e_1'\; ((\lambda v_1.\, e_2'\; ((\lambda v_2.\, \mathit{fst}\, k\; (v_1,\, v_2)),\; \mathit{snd}\, k)),\; \mathit{snd}\, k) \\
\textbf{try}\; e\; \textbf{with}\; \mathit{Exc}\, x \Rightarrow h &= \lambda k.\, e'\; (\mathit{fst}\, k,\; (\lambda v.\, h'[v/x]\; k)) \\
(\textbf{raise}\; \mathit{Exc}\;\; e)' &= \lambda k.\, e'\; (\mathit{snd}\, k)
\end{aligned}$$

Let's enrich this continuation calculus with stream codata. We must apply the CPS transformation to a function pair, $\langle f, g\rangle : \mathit{Stream}(A) \to A \times \mathit{Stream}(A)$, which implements the stream head and stream tail operations.

$$(\langle f, g\rangle)' = \lambda s.\, \lambda k.\, f'\; s\; ((\lambda v.\, g'\; s\; ((\lambda w.\, \mathit{fst}\, k\; (v, w)),\; \mathit{snd}\, k)),\; \mathit{snd}\, k)$$

Notice that in CPS-style expressions, every application of a function is a tail-call; i.e. it has no syntactic context in which it is expected to represent a value. This is guaranteed by the transformation rules. It is in this sense that the CPS transformation takes a lambda-calculus into an iteration calculus.

5. Coalgebras in Programming

A coalgebra consists of a carrier set, C, and a structure map $h : C \to F\, C$, where F is a functor that specifies the structure of the coalgebra. We shall refer to the family of coalgebras that have in common the functor F as the *co-variety of F-coalgebras.* As a simple example, suppose $F\, C = C \times C$. Then an example of an F-coalgebra is furnished by choosing $C = \mathit{int}$ and a structure map $h = \lambda n.\, (2n,\; 2n+1)$.

Of particular interest are the coalgebras of functors expressible in products, because then there are assured to exist F-dense sets in the category of "small" sets which we think of as containing realizable models for computation.

A particular F-coalgebra is characterized by its structure map. If F has the outer structure of a cartesian product, $F\ C = F_1\ C \times \ldots \times F_n\ C$, then the structure map will be a tupling operator, $h = \langle h_1, \ldots, h_n \rangle$, whose components are typed as $h_1 : C \to F_1\ C, \cdots h_n : C \to F_N\ C$.

For example, if $F\ C = A \times C$, then $h_1 : C \to A$ and $h_2 = C \to C$. This happens to be the co-variety to which the stream coalgebras belong. When the carrier is $\mathit{Stream}(A)$, the structure map is $\langle \mathit{Shd}, \mathit{Stl} \rangle$.

The importance of knowing that there are F-dense sets is that among the F-coalgebras there will then exist so-called *final* coalgebras. A final coalgebra exhibits the essential behavior common to every member of the co-variety without revealing extraneous detail, such as what is the type of its carrier and how are its witness functions implemented. We shall make this concept precise in the following discussion, but first, let's look at an example, characterizing a stream of ascending integer values as a coalgebra.

Example 5.1: Ascending integer values.
As the carrier of a coalgebra, we choose the type *int*. In this particular example, the type of the carrier and the type of stream elements is the same. The stream functor is then instantiated to $\mathit{int} \times \mathit{int}$. We choose as components of the structure function $h_1 = \mathit{id}_{\mathit{int}}$ and $h2 = (\lambda x.\, 1 + x)$.

The example above shows just one way to realize a stream of ascending integer values. A second way would be to program an integer stream generator in a more general representation, then specialize it to generate successive ascending values.

Example 5.2: A generator of integer values.
Let the carrier of a *Stream* coalgebra be the type $(\mathit{int} \to \mathit{int}) \times \mathit{int}$ and let the structure function have as its components $h_1 = \mathit{snd}$ and $h_2 = \langle \mathit{id}, \mathit{ap}_{\mathit{int}} \rangle$.

This stream generator can be specialized to generate any integer stream, each of whose values depends only upon the value immediately preceding it, by choosing an appropriate value of the carrier type. If we choose as the initial value of the carrier $((\lambda x.\, 1 + x),\ 0)$, then the generated stream will be the sequence of natural numbers in integer numeral representation.

5.1. Final coalgebras and information hiding

The co-data type $\mathit{Stream}(A)$ declared in Example 3.2.1 provides a generic type for any stream of A-typed elements. No implementation of the witness functions *Shd* and *Stl* is specified and thus the declaration provides no information about the underlying representation of a stream or how successive elements are calculated, whereas each of the two preceding examples was quite explicit about representation.

There is a unique, constructive mapping from the specification of an arbitrary stream of A-typed elements $C \xrightarrow{h} A \times C$ to an instance of the stream coalgebra $\mathit{Stream}(A) \xrightarrow{\langle \mathit{Shd}, \mathit{Stl} \rangle} A \times \mathit{Stream}(A)$. The construction is illustrated by the following diagram:

$$\begin{tikzcd}
A \times C \arrow[d, "id_A \times h'"'] & & C \arrow[ll, "\langle h_1, h_2 \rangle"'] \arrow[d, dotted, "h'"] \\
A \times \mathit{Stream}(A) \arrow[rr, "\mathit{StrCons}"] & & \mathit{Stream}(A)
\end{tikzcd}$$

where *StrCons* is merely a type coercion function that hides the stream representation. It satisfies the identity

$$\langle \mathit{Shd},\ \mathit{Stl}\rangle \circ \mathit{StrCons} = id_{A\times \mathit{Stream}(A)}$$

An F-coalgebra with the property that there is a unique mapping from any F-coalgebra into it, preserving the coalgebraic behavior, is said to be *final*. A final F-coalgebra is unique, up to isomorphism. A final coalgebra hides the representation and algorithmic implementation of the particular coalgebras of its co-variety.

Coalgebras have been studied as models for object-oriented programs in a series of papers by Bart Jacobs [Jac95, Jac96a, Jac96b].

5.2. A programming notation for coalgebras

It will be helpful to adopt an explicit notation for the use of coalgebras in programming, especially as no popular programming language has such a facility. The notation given here has been developed for the experimental programming language *DUALITY*.

A co-variety of coalgebras is declared by giving a signature that names its carrier and provides typed identifiers for its witness functions. Only the codomain types of the witness functions are given, as all share a common domain type, namely that of the carrier. The *DUALITY* notation provides for multiple sorts in a co-variety. We shall see an example of a multi-sorted co-variety later. For now, here's an example of the single-sorted co-variety *Stream*.

$$\textbf{cosig}\ \mathit{Stream}(A)\{\textbf{type}\ c;\ \$\mathit{shd} : A,\ \$\mathit{stl} : c\};$$

The identifiers with an initial "\$" symbol name the witness functions of the co-variety. Identifiers of the witness functions for the final coalgebra of the variety are derived from these identifiers by removing the "\$" symbol and capitalizing the initial letter.

To specify a pariticular coalgebra of a declared co-variety, bindings are given for the type parameter, the carrier type symbol and the witness functions. Thus we may specify a particular *Stream* coalgebra by the declaration

$$\textbf{coalgebra}\ \mathit{Ascending\_ints} = \mathit{Stream}(\mathit{int})\ \{c = \mathit{int};\ \$\mathit{shd} = \mathit{id},\ \$\mathit{stl} = (\lambda x.\, 1 + x)\};$$

The identifier *Ascending_ints* has been declared as a coalgebra, which is not a data or co-data type, but is a structural component that can be used with coalgebraic combinators to form data or co-data objects. The simplest coalgebraic combinator, *ana*—an abbreviation for *anamorphism* [MFP91], forms a *Stream* co-data object from a coalgebra and an initial value of the carrier type. We can use *ana* with the coalgebra *Ascending_ints* to form the stream of natural numbers

$$\mathit{Nats}\ :\ \mathit{Stream}(\mathit{int}) = \mathit{ana}[\mathit{stream}]\ \mathit{Ascending\_ints}\ 0$$

The effect of *ana* is to map the specified coalgebra, *Ascending_ints* to an instance of the final coalgebra *Stream*(*int*). This coalgebra specification is applied to 0, which is the initial value of the carrier. The codata object *Nats* can be accessed with the *Stream* witness functions. *Shd Nats* yields the integer 0, while *Stl Nats* yields the stream of ascending integers beginning with 1.

Example 5.2.1: Fibonacci sequence. The Fibonacci sequence generator produces the next value of the sequence by summing the two previous ones. This suggests that a carrier for a stream coalgebra might be a pair of integers. We can specify the sequence generator by

$$\begin{aligned}&\mathit{fibgen} = \mathit{ana}[\mathit{stream}]\ \{c = \mathit{int}\times\mathit{int};\ \$\mathit{shd} = \mathit{snd},\ \$\mathit{stl} = \langle \mathit{snd}, (+)\rangle\}\\ &\mathit{fibs} = \mathit{fibgen}\ (0,1)\end{aligned}$$

□

5.2.1. Stream representations in functional programming languages

Although familiar functional programming languages do not provide explicit notations for coalgebras, it is nevertheless possible to program coalgebraic functions with them. Let's take the *Ascending_ints* example and express it in two functional languages, Haskell and CAML.

Haskell is a functional language with lazy evaluation semantics. In Haskell, streams are most naturally represented as lists in which the computation of the list tail is suspended. Using this representation, we can make a recursive definition of *Ascending_ints* in Haskell as:

```
ascending_ints   :: int -> [int]
ascending_ints n = n : ascending_ints (1+n)

nats = ascending_ints 0
```

Since streams in Haskell do not have an abstract representation, we must specify witness functions of the appropriate types for the stream head and stream tail. The obvious choice is the functions `hd` and `tl` defined on the list data type.

CAML is a strict functional language, although it also has a stream co-data type.

```
>let rec ascending_ints n = [< 'n; (ascending_ints (1+n)) >];;
ascending_ints : int -> int stream = <fun>

>let nats = ascending_ints 0;;
nats : int stream = <abstr>
```

Streams in CAML are analyzed by pattern matches. To extract the head and tail of the stream defined above, we can write:

```
>let (h,t) = match nats with [< 'shd; stl >] -> (shd,stl);;
h : int = 0
t : int stream = <abstr>
```

However, if we should repeat the latter command, we would discover that the result of its second evaluation is

```
>let (h,t) = match nats with [< 'shd; stl >] -> (shd,stl);;
h : int = 1
t : int stream = <abstr>
```

Streams are implemented with mutable state variables in CAML. The operation of taking the stream tail updates the state of the stream implementation.

We can also define purely functional stream implementations in CAML, but in doing so, we shall lose the stream abstraction, just as in Haskell. In the following, the abstract type `int stream` is represented by the concrete type `(int -> int) * int`.

```
>let pr (f,g) x = (f x, g x);;
>let id x = x;;
>let ap (f,x) = f x;;

>let nats = ((function n -> 1+n), 0);;
>let shd = snd;;
```

```
>let stl = pr (id,ap);;
...
>shd nats;;
- : int = 0
>stl nats;;
- : (int -> int) * int = <fun>, 1
```

Another difference between the lazy list representation of streams in Haskell and the explicit, functional representation of streams in CAML is that previously computed values of stream elements are cached in a list data structure in Haskell. Caching of stream elements is sometimes useful, as programs may repeatedly access an initial segment of a stream. To provide caching of stream elements with the functional stream representation given above, it must be programmed explicitly. Here's an implementation for cached streams:

```
type 'a TaggedStream =
           Raw of ('a -> 'a) * 'a
         | Cached of ('a  * 'a CachedStream)
and  'a CachedStream = {Str : mutable 'a TaggedStream}
;;
let CoerceStream s = {Str = Raw s}
;;
let Cacheit =
 function {Str = Raw s} as cs ->
            cs.Str <- Cached (shd s, {Str = Raw (stl s)})
        | _ -> ()
;;
let chd = function {Str = Raw s} as cs ->
                           Cacheit cs;
                           (match cs.Str with
                               Cached (h, _) -> h)
                 | {Str = Cached (h, _)} -> h
;;
let ctl = function {Str = Raw s} as cs ->
                           Cacheit cs;
                           (match cs.Str with
                               Cached (_, t) -> t)
                 | {Str = Cached (_, t)} -> t
;;
```

The single mutable record field in an element of type *CachedStream* is modified by *Cacheit*, which substitutes an explicitly paired stream head and stream tail in place of a *Raw* stream. The pair is tagged so that it can be correctly analyzed by the witness functions *chd* and *ctl*.

5.2.2. Functions over streams

Having seen some examples of how to represent streams in a functional programming language, our next topic is to represent functions over streams. We shall consider three such: a function to retard a stream by one element; one that applies a given function to every element of a stream, and one that applies a stateful operation to the elements of a stream. With such a palette of stream-transforming functions, a great many interesting programs can be constructed.

Example 5.2.2.1: Delaying a stream.

Recalling that a stream is a sequence compatible with temporal ordering, to delay a stream, we need only to specify a new stream element at the front of the stream.

$$\begin{array}{l} \mathit{delay} \;:\; A \to \mathit{Stream}(A) \to \mathit{Stream}(A) \\ \mathit{delay}\ a\ s = \mathit{Stream}(A)\,\{c = s.c;\ \$\mathit{shd}\, s = a,\ \$\mathit{stl}\, s = s\}; \end{array}$$

□

Example 5.2.2.2: Mapping a function over the elements of a stream.

$$\begin{array}{l} \mathit{mapstream} \;:\; (A \to B) \to \mathit{Stream}(A) \to \mathit{Stream}(B) \\ \mathit{mapstream}\ f = \mathit{ana}[\mathit{stream}]\ \mathit{Stream}(B)\{c = B;\ \$\mathit{shd} = f \circ \mathit{Shd},\ \$\mathit{stl} = \mathit{Stl}\} \end{array}$$

□

Example 5.2.2.3: Mapping a stateful operator over a stream.
An abstract characterization of an event-driven transition system can be given by two functions, $g_{out} \;:\; \mathit{In\text{-}event} \to S \to \mathit{Out\text{-}event}$ and $\mathit{gnext} \;:\; \mathit{In\text{-}event} \to S \to S$, where *In-event* is a type of input events, *Out-event* is a type of output events and S is the type of a state variable.

$$\begin{array}{l} \mathit{mapstate} \;:\; (\mathit{In\text{-}event} \to S \to \mathit{Out\text{-}event}) \times (\mathit{In\text{-}event} \to S \to S) \to \\ \qquad\qquad\qquad\qquad S \times \mathit{Stream}(\mathit{In\text{-}event}) \to \mathit{Stream}(\mathit{Out\text{-}event}) \\ \mathit{mapstate}(g_{out}, g_{next}) = \mathit{ana}[\mathit{stream}]\,\{c = S \times \mathit{Stream}(\mathit{In\text{-}event}); \\ \qquad\qquad\qquad\qquad\qquad\qquad \$\mathit{shd} = \lambda(s, es).\, g_{out}\,(\mathit{Shd}\, es)\ s, \\ \qquad\qquad\qquad\qquad\qquad\qquad \$\mathit{stl} = \lambda(s, es).\,(g_{next}\,(\mathit{Shd}\, es)\ s,\ \mathit{Stl}\, es)\} \end{array}$$

□

Example 5.2.2.4: Verification condition for a stateful stream.
The coinduction principle for streams gives us immediately a verification condition for the stateful stream of Example 5.2.2.3.

$$\frac{\forall e : \mathit{In\text{-}event},\ s : S.\ (P(s) \Rightarrow Q(g_{out}(e, s))) \wedge (P(s) \Rightarrow P(g_{next}(e, s)))}{\begin{array}{l} \forall s_0 \;:\; S,\ es \;:\; \mathit{Stream}(\mathit{In\text{-}event}). \\ \qquad P(s_0) \Rightarrow \Box Q(\mathit{ana}[\mathit{stream}]\,\{c = S \times \mathit{Stream}(\mathit{In\text{-}event}); \\ \qquad\qquad\qquad\qquad\qquad \$\mathit{shd} = \lambda(s, es).\, g_{out}\,(\mathit{Shd}\, es)\ s, \\ \qquad\qquad\qquad\qquad\qquad \$\mathit{stl} = \lambda(s, es).\,(g_{next}\,(\mathit{Shd}\, es)\ s,\ \mathit{Stl}\, es)\}) \end{array}}$$

The predicate symbol P represents an invariant property of the state. The predicate symbol Q represents a property asserted of output events.
□

6. Searching and unbounded iteration

Streams are appropriate models for incremental, iterative computations, such as the activity of an operating system, an abstract machine, or a reactive controller. Streams are also appropriate models for potentially unbounded iterative computation over a sequence of values, such as linear search. In linear search, one can think of the succession of keys tested in the search as a stream, even if the keys are actually fetched from a data structure. In addition to specifying the stream to be searched, we need to specify the search criterion.

In contrast to an anamorphism, where the codomain is a type of codata, the codomain of a search function is the type of the result of the search, which we usually expect to be a data type. It is only in the algorithmic structure of the search function that we find its relationship to a co-variety. We call such a function a *hylomorphism* [MFP91].

Just as we saw that it was possible to specify a *Stream* anamorphism without the explicit use of recursion by using the *ana* combinator, a hylomorphism can also be specified without recursion. We need another combinator, which we call *cohom*, that takes a coalgebra specification and the specification of an analysis of values in the domain type to form an iteration control. The form is:

$$\mathit{cohom}[\mathit{sort}]\ \mathit{coalgebra\ spec\ coselector}$$

The *coselector* inherits its structure from the declaration of a co-variety. It is a functional form, taking an argument that has the carrier type of a coalgebra and yielding results expressed as applications of the coalgebra witness functions. A coselector for *Stream* co-data that tests whether a function $p : A \to \mathit{Bool}$ evaluates to tt when applied to the head of a stream is:

$$\textbf{coselect}\ \mathit{while}\ p = \mathit{Stream}(A)\,(\lambda x.\ \textbf{if}\ p\,(\$\mathit{shd}\,x)\ \textbf{then}\ \$\mathit{stl}\,x\ \textbf{else}\ \$\mathit{shd}\,x\ \textbf{fi})$$

With a function $r : A \to A$ we specify the *Stream* coalgebra

$$\textbf{coalgebra}\ \mathit{do}\ r = \mathit{Stream}(A)\{c = A,\ \$\mathit{shd} = \mathit{id}_A,\ \$\mathit{stl} = r\}$$

When used with *ana*, the coalgebra generates a stream of r-iterates:

$$\mathit{ana}[\mathit{stream}]\,(\mathit{do}\ r)\,a \rightsquigarrow a,\ r\ a,\ r(r\ a),\ r(r(r\ a)),\cdots$$

When the coalgebra and the coselector are combined with *cohom*, the test with the boolean function, p, is performed on the value of the carrier before each successive stream element is generated, and either the $\$\mathit{shd}$ witness function is applied to the carrier value, terminating the computation, or $\$\mathit{stl}$ is applied and the computation is iterated on the resulting carrier value. The combination defines the following function,

$$\textbf{while}\ p\ \textbf{do}\ r = \mathit{cohom}[\mathit{stream}]\,(\mathit{do}\ r)\,(\mathit{while}\ p)$$

whose verification condition is:

$$\frac{\forall x : A.\,(p\,x = \mathit{tt}) \Rightarrow P\,x \Rightarrow P(r\ x)}{\begin{array}{l}\forall x : A.\,P\,x \Rightarrow (p(\mathit{cohom}[\mathit{Stream}]\,(\mathit{do}\ r)\,(\mathit{while}\ p)\,x) = \mathit{ff})\\ \qquad\qquad \Rightarrow P(\mathit{cohom}[\mathit{Stream}]\,(\mathit{do}\ r)\,(\mathit{while}\ p)\,x)\end{array}}$$

Because iteration is controlled by a data-sensitive coselector, there is no bound on the number of iterations that *while p do r* may perform when applied to an argument value. The verification condition is a partial condition to conclude the property P holds of the result; an accompanying termination condition must be satisfied to conclude the result absolutely. We shall not explore termination conditions in these lectures.

The *while p* coselector is the hallmark of linear search. Other search algorithms can also be characterized by iterative, coalgebraic computations.

6.1. Binary search algorithms

Let's consider a co-variety with slightly richer structure than the *Stream* co-variety.

$$\textbf{cosig}\ \mathit{Btree}(A)\,\{\textbf{type}\ c;\ \ \$\mathit{val} : A,\ \ \$\mathit{left}, \$\mathit{right} : c\}$$

An example of a coalgebra specification in this co-variety is:

$$\begin{array}{l}\textbf{coalgebra}\ enum = Btree(int)\,\{c = int;\ \$val = id_{int},\\ \qquad\qquad\qquad\qquad\qquad\qquad \$left = (\lambda n.2n),\\ \qquad\qquad\qquad\qquad\qquad\qquad \$right = (\lambda n.2n+1)\}\end{array}$$

The positive integers, in binary representation, encode the co-data object generated by *enum* in an interesting way. Any access expression can be written as a composition of *$val* with a string of operators chosen from *$left* and *rgt*. Reading a composite access expression from left to right, if *$val* is interpreted as the binary digit 1, *$left* as binary 0, and *$right* as binary 1, the sequence of digits gives a binary representation of the value the access expression yields when applied to the co-data object $ana[Btree]\ enum\,1$.

Example6.1.1: Converting a bitstring to an integer. An algorithm to convert a finite list of binary bits to an integer value can be described with the coalgebra *enum*, or rather, of a related coalgebra of the same co-variety. The reason we shall need a related coalgebra is because the input value, the list of bits, must be incrementally analyzed, and the action of destructuring this input list must be accounted for in the witness functions of the coalgebra.

If the input list is *Nil*, the integer 0 will be the result. If the input list is nonempty, any initial segment of 0 bits must be discarded. We recognize the task of searching a list of unknown length for its first non-zero element as a linear search. Thus the outer structure of our algorithm will be:

$$\begin{array}{l}convert\ bs = \textbf{let}\ bs' = (\textbf{while case}\ bs\ \textbf{of}\\ \qquad\qquad\qquad\qquad\qquad\quad Nil \Rightarrow f\!f\\ \qquad\qquad\qquad\qquad\qquad\mid\ Cons(0,\_) \Rightarrow tt\\ \qquad\qquad\qquad\qquad\qquad\mid\ Cons(1,\_) \Rightarrow f\!f\\ \qquad\qquad\qquad\qquad\qquad \textbf{end do}\ tl)\ bs\ \textbf{in}\\ \qquad\qquad \textbf{case}\ bs'\ \textbf{of}\\ \qquad\qquad\quad Nil \Rightarrow 0\\ \qquad\qquad \mid\ \ \_ \Rightarrow convert'\ bs'\\ \qquad\qquad \textbf{end}\\ \qquad\qquad \textbf{where}\ convert' = cohom[btree]\ enum'\ discriminate\\ \qquad\qquad \textbf{where coalgebra}\ enum' = Btree(int)\,\{c = int \times list(bits);\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \$val = fst,\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \$left = \langle(\lambda n.2n),\ tl\rangle,\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \$right = \langle(\lambda n.2n+1),\ tl\rangle\}\\ \qquad\qquad \textbf{and coselect}\ discriminate = Btree(\lambda x.\ \textbf{case}\ x\ \textbf{of}\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad Nil \Rightarrow \$val\,x\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid\ Cons(0,bs) \Rightarrow \$left\,x\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid\ Cons(1,bs) \Rightarrow \$right\,x\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{end})\end{array}$$

As a recursive program, this could be written

$$\begin{array}{l}convert'\ x = \textbf{case}\ snd\,x\ \textbf{of}\\ \qquad\qquad\quad Nil \Rightarrow fst\,x\\ \qquad\qquad \mid\ Cons(0,bs) \Rightarrow convert'\ (2*(fst\,x),\ tl(snd\,x))\\ \qquad\qquad \mid\ Cons(1,bs) \Rightarrow convert'\ (2*(fst\,x)+1,\ tl(snd\,x))\\ \qquad\qquad \textbf{end}\end{array}$$

The witness functions of the coalgebra and the case analysis of the coselector have been combined, and the tail-recursive calls to *convert'* have been inserted. Although this is an intuitive way to write

the program, it is not so easy to reason about logically.
□

In other coalgebras of the *Btree* co-variety, the *\$left* and *\$right* operators may correspond to partitioning an interval of the real line, or to partioning an ordered set of keys such as words in a dictionary.

6.2. Reasoning about Btree coalgebras

We can give a coinduction rule for reasoning about coalgebras of the co-variety *Btree*. Unlike *Stream* coalgebras, it's not so easy to describe the totality of access paths in an infinite binary tree as in an infinite sequence. Where it was possible to use quantification of an integer index to describe iterations of the *\$stl* witness function, we now must describe all possible compositions of the two witness functions *\$left* and *\$right*.

Just such a quantification is provided in the branching-time temporal logic, CTL (*Computational Tree Logic*) [EC82] by the quantifier **AG**. With the aid of this quantifier, we can express the verification rule for *Btree* coalgebras:

$$\frac{\begin{array}{r}\forall x : S.\ \ P\,x \Rightarrow Q(f\,x)\\ P\,x \Rightarrow P(g\,x)\\ P\,x \Rightarrow P(h\,x)\end{array}}{\forall x_0 : S.\, P\,x_0 \Rightarrow \mathbf{AG}Q(ana[btree]\,\{c = S;\ \ \$val = f,\ \$left = g,\ \$right = h\}}$$

We can also give a verification condition for hylomorphisms of the *Btree* co-variety. For convenience, we suppose that the co-selector is defined using Dijkstra's **if ... fi** choice operator and that the guards are non-overlapping and cover the value domain S. That is, we assume that exactly one of the three guards p_0, p_1 and p_2 evaluates to *tt* for each value in the type S. Then the verification condition for *Btree*-hylomorphisms is:

$$\frac{\begin{array}{r}\forall x : S.\ \ P\,x \Rightarrow (p_0\,x = tt) \Rightarrow Q(f\,x)\\ P\,x \Rightarrow (p_1\,x = tt) \Rightarrow P(g\,x)\\ P\,x \Rightarrow (p_2\,x = tt) \Rightarrow P(h\,x)\end{array}}{\begin{array}{l}\forall x_0 : S.\, P\,x_0 \Rightarrow Q(cohom[btree]\,\{c = S;\ \ \$val = f,\ \$left = g,\ \$right = h\}\\ \qquad\qquad\qquad\qquad\qquad\begin{array}{rl}(\lambda x.\,\mathbf{if} & p_0\,x \Rightarrow \$val\,x\\ \mid & p_1\,x \Rightarrow \$left\,x\\ \mid & p_2\,x \Rightarrow \$right\,x\\ \mathbf{fi}) & x_0\end{array}\end{array}}$$

7. Multi-sorted coalgebras

Up to now, we have considered that an F-coalgebra is a structure $\langle A, h\rangle$, characterized by a single carrier type, A, and a structure map $h : A \to F\,A$. Suppose we generalize this somewhat. Suppose the carrier, A, is a product of n components, $A = A_1 \times \ldots \times A_n$; that the functor, F is also a product of n components, $F_1 \times \ldots \times F_n$; and the structure map is also an n-fold product, $h = h_1 \times \ldots \times h_n$, where $h_i : A_i \to F_i\,A$. This product structure allows for a situation in which the component structure maps may be mutually recursive.

To account for the component carriers and signatures of structure maps, they can be classified as belonging to distinct *sorts*. A sort is a name for a component of a co-variety. Multi-sorted algebras and coalgebras are not often considered in the realm of abstract mathematics, as there is no need to complicate theories with more detailed structure. In computer science, however, detail is unavoidable and we seek mathematical structures that can properly account for it.

As an example of a problem that can be characterized in terms of multi-sorted algebras, let's consider shift-reduce parsing of an LR(1) language. Shift-reduce parsers are finite transition systems capable of recognizing sentences of restricted subsets of context-free grammars. The control structure of a shift-reduce parser doesn't correspond directly to the structure of the grammar to be parsed, as it does for a recursive-descent parser. Rather, it corresponds to the finite-state control of a pushdown automaton derived from the grammar.

The structure of a non-backtracking shift-reduce parser is very much like that of any other reactive program. Actions of the parser are triggered by successive elements of an input stream of lexical tokens. The arriving tokens are its (token-valued) input events. The parser enters different states in its reactions to input events, and its reaction to a specific event is determined by its current state. The state includes a pushdown store in which a portion of the current state can be temporarily stored in one reaction and recovered in a later one.

Example 7.1: Shift-reduce parsing of a simple expression language.
To illustrate the technique, consider the following toy grammar for expressions.

G ::=	E '#'
E ::=	E + L
|	L
L ::=	'(' E ')'
|	*identifier*

We wish to translate sentences in the language defined by this grammar into an underlying *abstract syntax*, or term language. The term language is expressed as values in the datatype:

datatype *AST* = *Var* of *identifier* | *Add* of *AST* ∗ *AST*

A set of states is derived from an LR-grammar is characterized by a set of closed *configurations*. A configuration consists of a set of right-hand sides of productions and embedding a marker (called the parsing point) into each. A configuration is closed by adding to it the right-hand sides of all productions from any nonterminal that appears immediately to the right of the parsing point in any member of the configuration. In each right-hand side that is added to close a configuration, the parsing point is set to the left of its first symbol.

The initial configuration is that generated from the goal symbol by placing the parsing point to its left. New configurations are generated from existing ones by advancing the parsing point by one symbol in every member sequence, and calculating its closure. The generation of new configurations is assured to terminate because there are only finitely many productions in the grammar, and each of these is finite.

A configuration in which the parsing point lies to the left of a terminal symbol in some member sequence is called a *shift* state. A configuration in which the parsing point lies at the right end of a member sequence is called a *reduce* state. If both these situations occur in the same configuration, it is called a *shift-reduce* state.

We derive the following set of configurations.

$$
\begin{array}{llll}
S_0: & . \, E \, \# & S_5: & E \, \# \, . \\
 & . \, E \, + \, L & & \\
 & . \, L & S_6: & E + . \, L \\
 & . \, ident & & \quad . \, ident \\
 & . \, (\, E \,) & & \quad . \, (\, E \,) \\
 & & & \\
S_1: & E \, . \, \# & S_7: & (\, E \, . \,) \\
 & E \, . + L & & \quad E \, . + L \\
 & & & \\
S_2: & L \, . & S_8: & E \, + \, L \, . \\
 & & & \\
S_3: & ident \, . & S_9: & (\, E \,) \, . \\
 & & & \\
S_4: & (\, . \, E & & \\
 & \quad . \, E \, + \, L & & \\
 & \quad . \, L & & \\
 & \quad . \, ident & & \\
 & \quad . \, (\, E \,) & &
\end{array}
$$

States S_0, S_1, S_4, S_6, S_7 are shift states. S_2, S_3, S_5, S_8 and S_9 are reduce states. There are no shift-reduce states resulting from this grammar.

A context-free language differs from a regular language only because of the addition of matching bracket symbols. In the toy language of this example, the bracket symbols are left and right parentheses. At a left parenthesis, the parse is, in effect, restarted. At a right parenthesis, the nested parse is concluded and the one that was underway when the matching left parenthesis was encountered is resumed. Thus the only symbols at which the pushdown store comes into play are the left and right parentheses. At the left parenthesis, the state of the current parse is saved in the pushdown store and at the right parentesis, the saved parse state is popped from the pushdown store.

As an optimization, the action to be taken in a reduce state can be folded into the actions of the shift states on transitions that lead to that reduce state. Thus the states that need to be represented in a programmed implementation are just the shift states. In the following table, we show the shift states and indicate the state transitions to be taken on each terminal symbol. The push and pop actions on the pushdown store are also indicated. The names of the nonterminal symbols are not significant, once the configurations have been derived, and are represented by the symbol N in the configurations of the table.

$$
\begin{array}{llllll}
S_0: & . \, N \, \# & & S_6: & N + . \, N & \\
 & . \, ident & \to \, S_3 & & \quad . \, ident & \to \, S_3 \\
 & . \, (\, N \,) & \to \, S_4 & & \quad . \, (\, N \,) & \to \, S_4 \\
 & & & & & \\
S_1: & N \, . \, \# & \to \, S_5 & S_7: & (\, N \, . \,) & \to \, S_9 \\
 & N \, . + N & \to \, S_6 & & \quad N \, . + N & \to \, S_6 \\
 & & & & & \\
S_4: & (\, . \, N \,) & & & & \\
 & \quad . \, ident & \to \, S_3 & & & \\
 & \quad . \, (\, N \,) & \to \, S_4 & & &
\end{array}
$$

Notice that the transitions under terminals in states S_0, S_4 and S_6 are identical. These states can be merged. Notice also that no configuration element in these states has a nonterminal symbol immediately to the left of the parsing point. This indicates that the context to the left of the parsing point does not represent a complete term of an abstract syntax tree, but a partial term with a "hole" in it, expecting the continued parse to return a term with which to complete the partial term.

On the other hand, in the configurations of states S_1 and S_7, a nonterminal appears to the left of the parsing point in every sequence. In these states, the context represents a complete abstract syntax term and the continued parse will determine the use of the term. The states S_1 and S_7 do not contain identical configuration elements but the transitions on common terminal symbols are consistent. Thus these state also can be merged by taking a union of the configurations.

The merged states are $R_0 = S_0 \cup S_4 \cup S_6$ and $R_1 = S_1 \cup S_7$. Any transition from one of the states S_i to a reduce state can be represented in the merged states by a transition to R_1. A transition from a state S_i to a shift state can be represented by a transition to R_0. This leads to the characterization of the parser shown below:

$$\begin{array}{llll} R_0: & & & \\ & .\ \mathit{ident} & \to & R_1 \\ & .\ (\ N];) & \to & R_0/\mathit{push} \end{array} \qquad \begin{array}{llll} R_1: & & & \\ & N\ .\ \# & \to & \mathit{return}\ N \\ & (\ N\ .\ ) & \to & R_1/\mathit{pop} \\ & N\ .\ +\ N & \to & R_0 \end{array}$$

The characterization of the parser given above guides the declaration of a two-sorted co-variety. The sorts are $R0$ and $R1$, corresponding to the two states of the transition system.

$$\begin{aligned} \textbf{cosig}\ \mathit{E\_parse}(A)\{\textbf{type}\ b, c;\ & R0/b\,\{\mathit{\$ident}: A \to c,\ \mathit{\$lpar}: b,\ \mathit{\$err1}: \mathit{Ex}()\}, \\ & R1/c\,\{\mathit{\$plus}: b,\ \mathit{\$rpar}: \mathit{Ex}(c),\ \mathit{\$eof}: \mathit{Ex}(\mathit{AST}), \\ & \qquad \mathit{\$err2}: \mathit{Ex}()\}\} \end{aligned}$$

where Ex is a type constructor for the monad of exceptions. The result of evaluating an expression of type $Ex(A)$ is either a value of type A or is the result of raising an exception.

To complete the derivation of a parser as a transition system, we need to determine the type of the carrier of each sort. To generate an abstract syntax tree, the carrier type must represent the semantics of the token sequence parsed up to the current parsing point. For the $R0$ sort, it should contain a component of type AST; for the $R1$ sort, the component type is $AST \to AST$. Each carrier will also contain a component of type $\mathit{Stream}(\mathit{token})$ which represents the input to the parser.

In addition, the parser will use state. We need to specify the state type. Notice that a transition on a closing parenthesis is only allowed if there is a matching opening parenthesis in the left context. Also, a transition on the end symbol, '#', is allowed only if there is no unmatched opening parenthesis in the left context. Thus the state should include a component that determines whether or not there is an unmatched opening parenthesis in the left context of the parsing point. This state component needs to be saved in the pushdown store on a transition on an opening parenthesis.

In addition, the pushdown store must also contain a function of type $AST \to AST$ to represent the semantics of the contexts that precede stacked opening parentheses. We declare

$$\textbf{type}\ \mathit{State} = \{\mathit{Bkt}: \mathit{token};\ \mathit{Stack}: \mathit{list}(\mathit{token})\} \times (\mathit{AST} \to \mathit{AST}));$$

Next, we continue to elaborate the specification of a shift-reduce parser by declaring a two-sorted coalgebra. The witness functions of this coalgebra effects transformations of the state, as well as producing results to be output.

$$
\begin{array}{l}
\textbf{coalgebra } \mathit{shift} = E\_parse(\mathit{ident}) \\
\quad \{b = \mathit{State} \times \mathit{Stream}(\mathit{token}) \times (\mathit{AST} \to \mathit{AST}), \\
\quad\; c = \mathit{State} \times \mathit{Stream}(\mathit{token}) \times \mathit{AST}; \\
\quad\; R0\,\{\$\mathit{ident} = \lambda u.\, \lambda(S, ts, f).\, (S,\ \mathit{Stl}\, ts,\ ap(f,\ \mathit{Var}\, u)), \\
\qquad\quad \$\mathit{lpar} = \lambda(S, ts, f).\, (\{\mathit{Bkt} = \text{`)'};\ \mathit{Stack} = (S.\mathit{Bkt},\ f) :: S.\mathit{Stack}\}, \\
\qquad\qquad\qquad\qquad\qquad \mathit{Stl}\, ts,\ (\lambda x.\, x)), \\
\qquad\quad \$\mathit{err1} = \lambda X.\, \textbf{raise}\ \text{“Parsing error: expected `(' or identifier”}\}, \\
\quad\; R1\,\{\$\mathit{plus} = \lambda(S, ts, u).\, (S,\ \mathit{Stl}\, ts,\ (\lambda x.\, \mathit{Add}(u, x))), \\
\qquad\quad \$\mathit{rpar} = \lambda(S, ts, u).\, \textbf{if}\ S.\mathit{Bkt} = \text{`)'}\ \textbf{then}\ \{\mathit{Bkt} = \mathit{fst}(\mathit{hd}\, S.\mathit{Stack}), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Stack} = \mathit{tl}\, S.\mathit{Stack}\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Stl}\, ts,\ ap\,(\mathit{fst}(\mathit{hd}\, S.\mathit{Stack})), \\
\qquad\qquad\qquad\qquad\quad \textbf{else raise}\ \text{“Parse error: missing `)' ”}\,\textbf{fi}, \\
\qquad\quad \$\mathit{eof} = \lambda(S, ts, u).\, \textbf{if}\ S.\mathit{Bkt} = \text{`\#'}\ \textbf{then}\ u \\
\qquad\qquad\qquad\qquad\quad \textbf{else raise}\ \text{“Parse error: too many `)' ”}\,\textbf{fi}, \\
\qquad\quad \$\mathit{err2} = \lambda X.\, \textbf{raise}\ \text{“Parsing error: incomplete expr”}\}\}
\end{array}
$$

The last component of the coalgebraic specification is the co-selector:

$$
\begin{array}{rl}
\textbf{coselect } \mathit{checktoken} = R0\,(\lambda X. & \textbf{case}\ \mathit{Shd}\,(\mathit{snd}\, X)\ \textbf{of} \\
& \quad \text{ID}\ m \Rightarrow \$\mathit{ident}\, X\, m \\
& \mid\ \text{LPAREN} \Rightarrow \$\mathit{lpar}\, X \\
& \mid\ \_ \Rightarrow \$\mathit{err1}\, X \\
& \textbf{end}), \\
R1\,(\lambda X. & \textbf{case}\ \mathit{Shd}\,(\mathit{snd}\, X)\ \textbf{of} \\
& \quad \text{PLUS} \Rightarrow \$\mathit{plus}\, X \\
& \mid\ \text{RPAREN} \Rightarrow \$\mathit{rpar}\, X \\
& \mid\ \text{EOF} \Rightarrow \$\mathit{eof}\, X \\
& \mid\ \_ \Rightarrow \$\mathit{err2}\, X \\
& \textbf{end})
\end{array}
$$

The shift-reduce parser can then be constructed from these component specifications with the *cohom* combinator:

$$e\_parser = \mathit{cohom}[R0]\ \mathit{shift}\ \mathit{checktoken}$$

The parser applied to an initial state, token stream, and result function

$$(\{\mathit{Bkt} = \text{`\#'};\ \mathit{Stack} = \texttt{[]}\},\ \text{stdin},\ (\lambda x.\, x))$$

yields either a value in the type AST or an error message.
□

8. Conclusions

This paper provides a gentle, if incomplete, introduction to the use of coalgebraic structures in programming. The control structures gotten from coalgebras are iterative, rather than recursive. We have given some examples to demonstrate that iterative programs can also be functional, in the sense that there need be no hidden side effects. The intended effects of an iterative program are its visible reactions to events that occur in its environment.

A bonus for using a functional reactive style of programming is that one can reason about properties of a program in a logic that uses the programming language for its expressions, rather than reasoning about a semantics language in which the program is interpreted. We have indicated, again through examples, how detailed proof rules can be developed for co-recursive programs by analyzing the structures of codata objects.

Coalgebraic program structures are most commonly associated with the codata structure*Stream*. We envision a stream as a temporally evolving sequence of data values or objects. Traces of a finite state transition system fit this description. Coalgebraic structures for programming have begun to be studied seriously by a number of researchers. This approach is dual to the study of algebraic structures in programming, which led to a better understanding of recursive data types and the algorithms naturally associated with them. We anticipate that the study of coalgebraic techniques will similarly sharpen our understanding of codata and the iterative algorithms they naturally induce.

Bibliographie

[BB91] G. Berry and A. Benveniste. The synchronous approach to reactive and real-time systems. *Another Look at Real Time Programming, Proceedings of the IEEE*, 79:1270–1282, 1991.

[Ber89] G. Berry. Real-time programming: General purpose or special-purpose languages. In G. Ritter, editor, *Information Processing 89*, pages 11–17. Elsevier Science Publishers B.V. (North Holland), 1989.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[EC82] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronizations skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[EH97] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proc. of 1997 ACM/SIGPLAN International Conference on Functional Programming*, pages 163–173. ACM Press, 1997.

[Gor95] Andrew D. Gordon. Bisimilarity as a theory of functional programming. In *Proceedings of the Eleventh Conf. on Mathematical Foundations of Programming Semantics*, volume 1 of *Electronic Notes on Theoretical Computer Science.* Elsevier, March 1995. See also Andy's Web Page at http://research.microsoft.com/ adg/Publications/.

[HPE99] Paul Hudak, John Peterson, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In *Practical Aspects of Declarative Languages (PADL'99)*, volume 1551 of *Lecture Notes in Computer Science*, pages 91–105. Springer Verlag, 1999.

[Hud99] Paul Hudak. *The Haskell School of Expression.* Cambridge University Press, 1999.

[Jac95] Bart Jacobs. Mongruences and cofree algebras. In *AMAST'95*, volume 936 of *Lecture Notes in Computer Science*, pages 245–260. Springer Verlag, July 1995.

[Jac96a] Bart Jacobs. Coalgebraic specifications and models of deterministic hybrid systems. In *AMAST'96*, volume 1101 of *Lecture Notes in Computer Science*, pages 520–535. Springer Verlag, July 1996.

[Jac96b] Bart Jacobs. Inheritance and cofree constructions. In *ECOOP'96*, volume 1098 of *Lecture Notes in Computer Science*, pages 210–231. Springer Verlag, July 1996.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc. of 5th ACM Conf. on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, August 1991.

[MP91] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1991.