Productive Coprogramming with Guarded Recursion

Jeroen Slot en Ilse Pool

First slide

Last time we saw that corecursive functions have to be guarded. But guarded recursion does not always work, so we need something else, which we will be presenting. Since this can not be implemented in Coq, we will only use slides and mostly use Haskell notation.

Overview

Important from previous presentations

In the previous presentations co-recursion was introduced. We called the functions defined with `cofixpoint` the co-recursive functions.
These co-recursive functions have to satisfy the guard condition, meaning that its definition is only accepted if all recursive calls occur inside an argument of a constructor.
To check if the guard condition is satisfied, Coq uses guardedness checkers.
In this example we see that the definition of `repeat` is guarded because the recursive call occurs inside an argument of the constructor `Lcons`.

Productivity

We introduce a new notion, namely that of productivity. For a program to be productive means that even if it generates an infinite amount of data, each piece will be generated in finite time. To ensure that a recursively defined program is safely used in a total setting, the system must ensure that all recursive definitions are productive.
It holds that productivity is guaranteed by guardedness.
In this presentation we will often use StreamCons in the examples, which is a constructor. So we use these notions (point at the slide).

Examples

We will now show some examples of recursive definitions that are guarded and productive or not.

ones defines a stream of ones. It is an example of a guarded definition, because ones is only called in an argument of the constructor StreamCons. The definition produces an infinite stream, but each piece is delivered to us in finite time, so ones is productive.
As you may recall from the previous presentations, **filter** takes from a stream only those elements that satisfy the boolean predicate f. And you may also recall that this definition was not guarded, because the recursive call in the else-statement does not occur in an argument of a constructor. To see if filter is productive, we look at the case where elements of the stream are filtered out. Then it will not return anything, so it is not productive.

Why guardedness checks are not enough - 1

In the examples of ones and filter, guarded recursion is helpful, but we will now see an example of a definition in which it is not.
Mergef takes a function two streams, and essentially just wants to merge those two streams. A guardedness check would rightfully reject it, since the recursive call of mergef does not occur in a constructor. If we take a specific function *badf* as on the slide, mergef will hang on all pairs of streams, and thus it wil not be productive.

Why guardedness checks are not enough - 2

However, there are functions so that mergef is productive, but a guardedness checker does not allow us to express this. Therefor we want to make f return instructions on how to transform the stream back to mergef, which then executes them in a way that the guardedness checker can see it is guarded. Essentially this means that we want to change the type of f so that mergef will be guarded.
Examples of good functions are these (point at the slide). The first one allows for merging functions to operate element-wise. The second one allows for the functional argument to replace each pair of elements from the input streams with a non-empty list of values.
But it is difficult to see if we can capture all possibilities for good functions in a single type.

Solution: a guardedness type constructor

We introduce a guardedness type constructor that may only be used in a guarded way.
We write it with a triangle, like you can see on the slide. A useful way to think about $\triangleright$A is as a value of A that is only available "tomorrow".
We need a constructor to brigde the gap between "tomorrow" and "today".
We can now change the type of f into (point at slide).

Guardedness type constructor

To be able to introduce guarded types into the system, we must alter the type of StreamCons as follows (point at slide). StreamCons now takes a guarded stream "today" and produces a non-guareded stream "tomorrow".
deStreamCons is the inverse operation that takes a full stream and returns the integer and the guareded remainder of the stream.

fix, pure and $\circledast$

We define the fixpoint-operator like this. We can, for example, define *ones* with **fix**.
We also define the operators **pure** and $\circledast$.
These operators are useful, for example for making a productive *mergef*.

Productive *mergef*

Using the guardedness type constructor, **fix** and $\circledast$ we can make a *mergef* that is productive.

From the infinite to the finite

The constructor $\triangleright$ replaces the syntatic guardedness that can be found in programs like coq with a compositional type-based one. We now describe a problem with guarded recursion when attempting to combine infinite and finite data. We propose a solution with clock variables, which we discuss after this example.
Consider the *take* function that takes a finite prefix of an infinite stream, and creates a finite list of it. It has the following type (point at the slide). We run into a difficulty, namely we want this variable s' (point at the slide), which we have obtained from deStreamCons, to have have type Stream, but it has type $\triangleright$ Stream.
Intuitively we can look at it like this: the guarded type constructor slices the construction of the infinite stream into discrete steps. But when the stream is fully constructed, it should not matter when each part was constructed, but here it still does. We fix this by introducing so called clock variables.

Clock variables - 1

A clock variable κ represents an individual time sequence that can be used for safe construction of infinite data, like streams. We write $\triangleright^{\kappa}$A to indicate which time stream we are considering, which time stream they are constructed on.
We define the constructor and deconstructor with clock variables like this (point at slide)

Clock variables - 2

We now regard the type guarded stream kappa as the type of infinite streams in the process of construction. A finished infinite stream is represented with the universal quantifier over kappa, so for all kappa guarded stream kappa.
We define **fix**, **pure** and $\circledast$ with clock variables as on the slide.
We also introduce another function, **force**, which forces a value that, for any time stream, is one time step away, to be available instantly.

Example force

In the last line, after force, we get something of type $\forall\kappa.\triangleright^{\kappa}\text{GStream}^{\kappa}$, but we need something of type $\forall\kappa.\text{GStream}^{\kappa}$, so **force** is needed here.

Well-typed *take*

Now we have the clock variables, we can make the well-typed function *take* that we desire, which takes a finite observation of a piece of infinite data.

(Final) Coalgebras

Inductive types define an algebra. Now if we were to look at a collection of coinductive types, we obtain a coalgebra. We can now use clock variables to define the *unfold* combinator in terms of the **fix** operator, and StreamCons. With the *unfold* combinator we obtain a final coalgebra from the coalgebra of coinductive types. This is useful when observing constructions like *take*, which take a infinite object, and observers a finite part of that object. These (final) coalgebras give us some categorical tools for handling clock variables and Streams.

Nats and Naats

We end on an example to show the use of clock variables.
They allow us to make distinctions between functions that are extensionally equal, but differ in their productivity. In plain Haskell, the following two stream mapping functions have the same behaviour on any completely constructed stream. The first *map* processes each element one at a time (point at the slide) while *maap* processes elements two at a time by pattern matching. If all the elements of the input are available at once, and not say one today and the other tomorrow, then they have the same behaviour. However, if these functions are passed streams that are still being constructed, then their behaviour can differ significantly. Consider these two definitions of a stream of natural numbers, using *map* and *maap* (point at the slide). We see that *nats* produces the infinite stream of natural numbers, but *naats* produces nothing. The function *maap* expects two elements to be available as input, while *naats* has only provided one. We can stat their different behaviour

using the following types in our system (points at the slide). Using this type theory, the previous given definition of *naats* will be rejected by our system.