Individual Systems Practical

November 14, 2016

1 The two cases of an algorithm

With regard to the design of the Haskell framework, we need to find a systematic way of:

1. Composing basic algorithms with control algorithms, e.g. in the case of dead-lock detection and termination detection. Should these control algorithms work on any *generic* basic algorithm, or should the basic algorithm be classified as a suitable algorithm for applying one of the control algorithms on top?

2. Recomputing an execution in the case of a reordered event. Suppose that some execution is modified by the student by swapping two events (that are considered swappable, see the other document), then the consequence of the swapping gesture is the recomputation of all future events. Note that swapping two concurrent events does not require the recomputation of all future events, and swapping two causally related events does require such recomputation. How can we best model all possible computations?

Let us first tackle the first question. A first possibility was already explored during the non-functional prototype, which is rather crude: any process that waits for receiving a message is considered passive. If you look at Chapter 6, Termination Detection, the first diagram there shows two states a process can be in: *active* and *passive*. As is clearly shown there, it is possible to remain active while waiting for receiving a message! Only a special internal event signals that a process is deemed passive. And after receiving a message, the process automatically becomes active again.

It is conceivable that we can impose this state machine onto any process, if we simply disregard the receive event that remains at an active process. The construction (or better, transformation) can also apply to any generic basic algorithm. We can inspect any process in conjunction with its history and say it is passive only if it is in a state that waits for a message where its preceding state is a special "become passive" internal event. Now, to transform any generic basic algorithm into an algorithm which also has this "become passive" internal event, we simply insert this internal event before every message it waits for to receive. Any algorithm that wants to remain active has to implement the

"become passive" internal event by itself, and to become an instance of suitable basic algorithms it only has to say which internal event should be considered as the special "become passive" internal event. This suggests that we should implement these algorithms as typeclasses in Haskell. Even if we were to decide that any generic basic algorithm should be usable, we simply provide a translation function from any algorithm that inserts the special event before each receive, that is an instance of this typeclass.

Now for the second question, we have found that a list of events can be useful. In the prototype, a simulation is a list of events, where events are either send, internal or receive. The list determines the relative order of the events. Consider how the glue code would allow students to reorder events: an algorithm is simulated by recursively producing a list of events. The list of events is translated into Java objects that implement a Java model for use in the user interface. Then the student reorders an event, and the new order is translated back to a list of all events preceding and including the swapped event. Then the algorithm is simulated by producing the rest of the list.

A different implementation would be to build a tree, where each branch represents a different order of events. We again have a list of events, now as a path back to the root of the tree. This is has an advantage compared to translating back to a list, since the tree already contains all possible orderings of events as different paths. Reordering two events is then simply taking a different branch from this tree, and computing the remaining path. The only thing that has to be translated back is where the branch happens in the tree. Since the tree is evaluated lazily in Haskell, it has no additional performance overhead compared to the list approach.

Considering that in the future we might also want to model probabilistic algorithms, the tree can then also contain branches in the place of random choices. The student can now explore the branches by influencing the outcome of a random process, by basically selecting which branch will be taken. To model probabilistic algorithms using lists is harder, since it involves the creation of possible outcomes. However, in the case of Byzantine processes, where a student can supply its own messages into the process, we already have a need to allow the construction of arbitrary messages from the user interface and the injection of these messages into the simulation.