Laziness and Declarative Concurrency

Raphaël Collet
Université Catholique de Louvain,
B-1348 Louvain-la-Neuve, Belgium
raph@info.ucl.ac.be

May 7, 2004

Abstract

Concurrency and distribution in a programming language are simple to deal with when they are not mixed with stateful operations. Threads and processes can communicate with dataflow variables, which makes synchronization implicit. The multiparadigm programming language Oz provides such a model, together with a fine integration of stateful entities.

We have extended the concurrent declarative framework of Oz with laziness, or by-need synchronization. This extension provides a simple yet powerful computation model, which can be implemented efficiently.

1 Introduction

Distributed programs are concurrent by nature. Writing correct distributed programs is as difficult as writing correct concurrent programs. And this task is often considered hard and painful. This comes from the difficulty to identify critical sections in a program, during which two concurrent components may change a shared state. Those concurrent accesses make the state of the program difficult to reason with.

A lot of solutions have been proposed, such as semaphores, monitors, message-passing, etc. All of them address the problem of concurrent accesses to a shared state. Though such situations are unavoidable, lots of interactions between concurrent components of a program do not require a shared state.

Let us illustrate this with an example. Imagine two processes, one producer sending items to a consumer through a buffer. The items are consumed in order. As both processes access the message buffer concurrently, the implementation of the buffer generally contains critical sections. This is because the buffer is shared and stateful. Though the whole sequence of transmitted items does not change over time. Therefore this system is fundamentally stateless.

To design this system as a declarative system, the producer and the consumer simply share the list of produced items. Remember that the list is a constant. But the whole list doesn't need to be known at once. The trick is to allow the

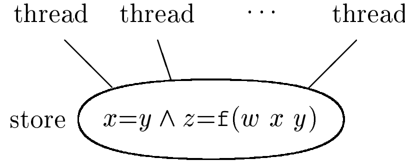

Figure 1: The execution model of Oz

tail of this list to be *unknown* until the producer makes the next item. If the consumer encounters an unknown value, it blocks until it becomes known. This behavior is called *dataflow*, and is completely stateless. Therefore declarative concurrent programs are much simpler to design, and to reason about.

The paper presents a declarative concurrent language, together with its computation model. We then give a small extension to the language for writing transparent lazy computations. We finally argue why imperative languages may benefit from this model.

2 Declarative Concurrency

The model. We present a model for declarative concurrency by way of the multiparadigm language Oz. The full model of the language is given in the book [4]. For the purpose of this paper, we restrict ourselves to its concurrent declarative part. The computation model of Oz is close to Saraswat's concurrent constraints model [2]. As depicted in Fig. 1, a program consists of several threads of statements executing concurrently and communicating via a shared store [3]. The threads access the store by logic variables.

The store consists of a conjunction of constraints over logic variables. The constraints are logic predicates that restrict the values of the variables. The main constraint we consider here is equality on logic variables and records, whose fields are themselves logic variables. The store provides the reflexivity, symmetry and transitivity of equality. More details can be found in [4].

The store has only two operations: *ask* and *tell*. Telling a constraint c simply adds it to the store. Asking a constraint c waits until the store can logically infer c or $\neg c$. No information can be removed or changed. The store thus evolves *monotonically*, guaranteeing the declarative nature of the language.

The language. The syntax of an Oz statement is given in Fig. 2. The statements **`skip`**, S_1 S_2, and **`thread`** S **`end`** respectively denote the empty statement, the sequence, and thread creation.

The **`local`** statement creates a new logic variable in the store, denotes it with an identifier, and defines its lexical scope. The statements $X{=}Y$ and $X{=}f(Y_1 \cdots Y_n)$ unify their terms by telling the necessary constraints in the store. If the constraints make the store inconsistent, the program fails. The

$$
\begin{array}{rcl}
S & ::= & \texttt{skip} \mid S_1\ S_2 \mid \texttt{thread}\ S\ \texttt{end} \\
& \mid & \texttt{local}\ X\ \texttt{in}\ S\ \texttt{end} \\
& \mid & X{=}Y \mid X{=}f(Y_1 \cdots Y_n) \\
& \mid & \texttt{if}\ X\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end} \\
& \mid & \texttt{case}\ X\ \texttt{of}\ f(Y_1 \cdots Y_n)\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end} \\
& \mid & \texttt{proc}\ \{P\ X_1 \cdots X_n\}\ S\ \texttt{end} \mid \{P\ X_1 \cdots X_n\}
\end{array}
$$

Figure 2: Syntax of kernel statements of Oz

constraint system is chosen such that the inconsistency can be detected immediately.

The conditional **if** X **then** S_1 **else** S_2 **end** blocks until the variable X is determined by the store. If X is equal to **true** (resp. **false**), the statement continues with S_1 (resp. S_2). The **case** statement is similar, except that X must be sufficiently constrained to match the given pattern. If the pattern matches, S_1 is executed with the identifiers Y_i denoting the corresponding fields in X. Otherwise, S_2 is executed.

The **proc** statement creates a closure named ξ in the store, and reduces to $P{=}\xi$. The procedure application $\{P\ Y_1 \cdots Y_n\}$ blocks until P is bound to a closure ξ, then reduces to S, with the Y_i's substituted for the X_i's. As procedures appear as values, the language is higher-order.

An example. Figure 3 presents the code to implement the introductory example, with a producer and a consumer. Both execute in a separate thread. The code appears twice. The left version has been written in the full Oz language, which provides functions and expressions. Note that the bar operation `X|Y` is syntactic sugar for the tuple `bar(X Y)`. The right version is the expansion of the left one in the kernel language.

The confluence property. In the absence of nondeterministic statement, the monotonicity of the store makes every program deterministic. The order of executed statements may vary from one execution to an other, but the final state of the program is unique[1]. This property, defining a concept of determinism for concurrent programs, is called *confluence*.

3 By-need Synchronization

Programming a demand-driven computation in a dataflow concurrent language is easy. Simply put the computation in a thread that waits until another thread manifests its need for the result.

[1]The final state of the program comprises both the constraint store and the remaining blocked threads.

```
fun {Produce State}
   State|{Produce State+1}
end

proc {Consume In State}
   case In of Item|In2 then
      {Consume In2 State+Item}
   else skip end
end

{Consume
 thread {Produce 1} end
 0}
```

```
proc {Produce State Out}
   local State2 Out2 in
      State2 = State + 1
      Out = State|Out2
      {Produce State2 Out2}
   end
end

proc {Consume In State}
   case In of Item|In2 then
      local State2 in
         State2 = State + Item
         {Consume In2 State2}
      end
   else skip end
end

local S in
   thread {Produce 1 S} end
   {Consume S 0}
end
```

Figure 3: A declarative producer-consumer system.

```
thread {Wait Trigger} {Compute Result} end
```

The procedure `Wait` simply blocks until its argument is bound to a value in the store. The variable `Trigger` is then bound by demanding components.

```
Trigger = unit
```

The problem of this scheme is that the demand must be explicit. So turning an eager computation into a lazy one requires the programmer to change all the components using its result. The model we developped avoids this problem by making the demand implicit.

The model. The lazy computation should block until an information in the store reflects the demand. We model this information as a constraint on a variable. Given a variable x in the store, the constraint need(x) signals that the value of x is needed by a component. The demanding component then *implicitly* tells this constraint to the store.

The constraint need(x) does not restrict the possible value of x. It is logically equivalent to **true**. The usual rule of substitution by equals applies to it, i.e., if the store infers need(x) and x=y, then it also infers need(y). Finally, to ensure the monotonicity of the store, we state that *values* are always needed. In others words, if the store infers x=v for a value v, then it also infers need(x).

Language extension. The statement {`WaitNeeded` x} blocks until the store infers need(x). This allows to program the lazy computation. One typically

```
fun lazy {Produce State}
   State|{Produce State+1}
end
```

```
proc {Produce State Out}
   thread
      {WaitNeeded Out}
      local State2 Out2 in
         State2 = State + 1
         Out = State|Out2
         {Produce State2 Out2}
      end
   end
end
```

Figure 4: A lazy version of the producer.

creates a thread for this purpose. On the "other side", any statement that blocks on a variable x automatically tells need(x) to the store[2]. For instance, the statement **case** X **of**... automatically needs X. This makes the demand implicit. An explicit demand is made by creating a thread blocking on the variable.

A lazy producer with a bounded buffer. We can improve our example above by making the producer lazy. It then synchronizes on the demand to avoid a possible overloading of the consumer. No modification of the consumer is needed. The modified code of the producer is shown in Fig. 4. In the left version, the full language provides the annotation `lazy` for lazy functions.

One can even improve the responsivity of the producer, by forcing it to produce a few items in advance. A bounded buffer can be used for this purpose. The code is given in Fig. 5. This is extremely useful if the producer and the consumer are on different machines, and communicate through a network. The programmer then chooses the size of the buffer to make the network delays unnoticeable.

4 An Opportunity for Imperative Languages

Imperative languages have hard to deal with concurrency. The main reason being that the communication between concurrent components is stateful. Complex primitives exist to avoid state inconsistencies due to concurrent state updates. Laziness in such languages is seldom. Because of the omnipresence of side effects, it often breaks the modularity of the program.

Though, adding dataflow concurrency is possible. The language Flow Java conservatively extends Java with single assignment variables and futures as variants of logic variables [1]. Just like in Oz, a thread trying to use the value of a single assignment variable automatically blocks, and become runnable again as soon as the value is available. This language gives the opportunity of writing

[2]For the sake of simplicity, we do not consider threads that can block on several variables. For a precise definition, consult [4]

```
fun {Buffer In N}
   End=thread {List.drop In N} end
   fun lazy {Loop In End}
      case In of I|In2 then
         I|{Loop In2 thread End.2 end}
      end
   end
in
   {Loop In End}
end

{Consumer {Buffer {Producer 1} 5} 0}
```

Figure 5: A bounded buffer of size 5, and how to use it.

declarative concurrent components in Java. Adding by-need synchronization in this language is possible, using the same model as in Oz.

5 Conclusion

Declarative concurrency is a simple computation model, that allows highly concurrent components, without the nondeterminism inherent to stateful concurrent components. Declarative components are often simpler, and much easier to reason about. The programming language Oz provides declarative concurrency as part of a whole model. This gives a solid foundation for truly multiparadigm programming.

We have extended the declarative concurrent model with by-need synchronization, which is used as a building block for concurrent lazy computations. The extension fits extremely well with the model, and has the advantage that the demand is implicit. One can thus incorporate laziness in a component without changing the components that use it.

References

[1] Frej Drejhammar, Christian Schulte, Seif Haridi, and Per Brand. Flow Java: Declarative concurrency for Java. In Catuscia Palamidessi, editor, *Proceedings of the Nineteenth International Conference on Logic Programming*, volume 2916 of *Lecture Notes in Computer Science*, pages 346–360, Mumbai, India, December 2003. Springer-Verlag.

[2] Vijay A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[3] Gert Smolka. The Oz programming model. In *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[4] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.