On Futures for Streaming Data in ABS
(Short Paper)

Keyvan Azadbakht(✉), Nikolaos Bezirgiannis, and Frank S. de Boer

Centrum Wiskunde & Informatica (CWI), Amsterdam, Netherlands
{k.azadbakht,n.bezirgiannis,f.s.de.boer}@cwi.nl

Abstract. Many modern distributed software applications require a continuous interaction between their components exploiting streaming data from the server to the client. The Abstract Behavioral Specification (ABS) language has been developed for the modeling and analysis of distributed systems. In ABS, concurrent objects communicate by calling each other's methods asynchronously. Return values are communicated asynchronously too via the return statement and so-called futures. In this paper, we extend the basic ABS model of asynchronous method invocation and return in order to support the streaming of data. We introduce the notion of a "Future-based Data Stream" to extend the ABS. The application of this notion and its impact on performance are illustrated by means of a case study in the domain of social networks simulation.

Keywords: Future · Streaming · Cooperative scheduling · Active object · Programming language · Social network

1 Introduction

Streaming data is a client/server pattern used in many distributed applications. It consists of a continuous generation of data by the server where the generated data is processed by the client sequentially and incrementally. The Abstract Behavioral Specification (ABS) language [1] has been developed for formal modeling and analysis of distributed systems. In ABS, concurrent objects represent processes that execute in parallel and interact via asynchronous communication of messages. A message specifies a call to one of the methods of the called object. Return values are communicated asynchronously via so-called futures [2]. Futures are dynamically generated references for checking the availability of the return value and its reading.

In this paper, we extend this basic model of asynchronous method invocation and return to support the streaming of data between the server and the client. We introduce the notion of "Future-Based Data Stream" by extending the syntax with a *yield* statement which returns a value specified by its argument *without* terminating the execution of the method, which thus proceeds as specified. The generated values can be obtained incrementally and sequentially by querying the

A. Bouajjani and A. Silva (Eds.): FORTE 2017, LNCS 10321, pp. 67–73, 2017.
DOI: 10.1007/978-3-319-60225-7_5

future corresponding to this method invocation. The standard return statement terminates the method execution and is used to signal the end of the generated stream of data.

As a proof of concept, we also present the impact of the above-mentioned feature on performance in the implementation of distributed application for the generation of social networks.

Related Work. There are different programming constructs for streaming data. *Asynchronous generators* as specified in [3], enable the streaming of data in an asynchronous method invocation. This includes, on the callee side, yielding the data, and on the caller side receiving them as an asynchronous iterator or raising an exception if there is no further yielded data. These generators are defined in the context of the multithreaded model of concurrency, where asynchrony are provided by spawning a thread for a method call. *Akka Streams* [4] provide an API to specify a streaming setup between actors which allows to adapt behavior to the underlying resources in terms of both memory and speed. There are also languages which utilize the notion of a *channel* as a means of communication, inspired by the model of *Communicating Sequential Processes* (CSP). For instance, *JCSP* [5] is a library in Java that provides CSP-like elements, e.g., processes and channels with read and write possibilities.

Similarly to asynchronous generators, as proposed below streaming data is fully integrated with asynchronous method invocation, i.e., it is not a separate orthogonal concept like channels are. But its integration with the ABS language allows for an additional loose coupling between the producer and consumer of data streams: by means of cooperative scheduling of tasks the consumption of data can be interleaved with other tasks on demand. Moreover, the notion of data streaming abstracts from the specific implementation of ABS. In our case, we make use of the distributed Haskell backend of ABS [6] for the case study on future-based data streams reported in this paper.

This paper is organized as follows: a brief description of the ABS programming language is given in Sect. 2. The notion of Future-Based Data Stream is specified as an extension of ABS in Sect. 3. In Sect. 4, a case study on social network simulation is discussed, which uses the proposed notion of streams. Finally we conclude in Sect. 5.

2 The ABS Programming Language

Here we briefly highlight the main features of ABS relevant to our work in this paper. In ABS, parallel (or concurrent) processes are generated by asynchronous method calls of the form $f = o!m(\bar{e})$, where f is a future used as a reference to the return value of the asynchronous call of the method m, o is an expression denoting the called object, and $\bar{e}$ are the actual parameters. Such a call generates a process for the execution of the invoked method in the (unique) *concurrent object group* (cog) to which the called object belongs. Within such a group at most one process is executing. The executing process of a cog is however executing in parallel with all the executing processes of the other groups. A cog

is created by the expression **new** C, where C denotes a class. The statement **new local** C adds a new instance of class C to the group of the object that creates this instance.

Further, ABS features synchronous method calls and a high-level synchronization mechanism (i.e., cooperative scheduling) which allows a cog to suspend the execution of the current process and schedule another (enabled) process for execution, by means of *await* and *suspend* statements. The *await f?* suspends the current process of the active object if the future f is not resolved yet, or it skips otherwise. The *await b* similarly suspends the current process if the boolean guard b evaluates to *false*. Finally, the *suspend* statement blocks the current process unconditionally. The process to be activated next is selected in a cog based on the scheduling policy. The *f.get* also queries the resolution of the future f and reads its value if it is resolved or it blocks the whole cog otherwise.

3 Future-Based Data Streams

In this section, future based data streaming is specified in the context of the ABS language which exploits the notion of cogs and cooperative scheduling.

The simple code example in Fig. 1 illustrates how data streams can be used in an ABS program extended by this feature. The program is comprised of two classes `Producer` and `Consumer` which implement the `IProd` and `ICons` interfaces, respectively, followed by the main block of the program. First, the runtime system instantiates two cogs, each of which contains one active object, i.e. the objects `prod` and `con`. The main cog then calls asynchronously the `process` method of the object `con`. The `process` method calls asynchronously the method `primes` which basically generates the first n prime numbers. The `primes` method is a *streamer method*, that is, its return type is `Str<Int>` and the method specification is allowed to contain the `yield` statement which is, roughly speaking, a non-terminating `return` statement.

Therefore, the prime numbers generated by the `primes` are streamed to the caller via a data stream. The last value by `return` statement is followed by a special token (e.g., ***eof***) to state that there is no further value to be streamed. On the caller side, the return values are assigned to the variable `r` which is a stream buffer enabling the above-mentioned streaming of return values from the callee to the caller process. The `StrFut<T>` can only type a variable which is assigned by an asynchronous method call where the callee is a streamer method returning values of type `T`.

The `awaitAll` statement of the `Consumer` class is a mechanism to retrieve all the prime numbers from the stream `r`. Based on the state of the stream, the statement behaves in three different ways: (1) if there is at least one value in the stream then the first value will be retrieved, assigned to x, and removed from the stream buffer. The following block of the statement will be also executed. This will repeat until the buffer is empty. (2) If there is no value in the buffer but there may be more values to be received (i.e., ***eof*** is not buffered yet), then the process cooperatively releases control so that another process of the `consumer` is

activated. The process will be activated later once the stream is not empty. (3) If there is no value in the buffer and the ***eof*** is in the stream then the statement will skip.

```
interface IProd {
  Str<Int> primes(Int n);
}

class Producer implements IProd
{
  Str<Int> primes(Int n) {
    Int i = 1;
    Int j = 2;
    while(i<n)
    {
       yield j;
       j = nextPrime(j);
       i = i + 1;
    }
    return j;
  }

  Int nextPrime(Int x) {
    // return the smallest prime > x
  }
}
```

```
interface ICons {
  Unit process(IProd o);
}

class Consumer implements ICons
{
  Unit process(IProd o){
     StrFut<Int> r = o!primes(1000);
     awaitAll r? in x
     {
       // process x
     }
  }
}
// main block
{
  IProd prod = new Producer();
  ICons con = new Consumer();
  con ! process(prod);
}
```

Fig. 1. An example in ABS extended by future-based data streams

Syntax. The syntax of our proposed extension of ABS, i.e., that of future-based data streams, is specified in Fig. 2. The only syntactic extension on the callee side is the **yield** statement, that can only be used in the specification of a *streamer* method. The rest of the statements are related to the specification of a caller to a streamer method.

awaitAll is already described in the above example. The **await-catch** statement is a single-fetch version of the repetitive **awaitAll**: (1) If there is at least one value in the stream buffer then it retrieves the head of the buffer and assigns it to x. It also removes the value from the buffer, but not repetition takes places. (2) As before, if there is no value in the buffer then the process releases the control cooperatively. (3) If there is ***eof*** in the stream buffer then, deviating from **awaitAll**, the statement s is executed. The **getAll** and **get-catch** coincide semantically to the **awaitAll** and **await-catch** respectively, except for releasing control, where the whole cog is suspended rather than the current process.

$$s ::= \textbf{yield}\ e \mid \textbf{awaitAll}\ e?\ \textbf{in}\ x\ \{s\} \mid \textbf{await}\ e?\ \textbf{in}\ x\ \textbf{catch}\ \{s\} \mid e.\textbf{getAll in}\ x\{s\} \mid e.\textbf{get in}\ x\ \textbf{catch}\ \{s\}$$

Fig. 2. Syntax

4 Case Study

Simulation of massive social networks is of great importance. Typically, larger networks are structurally different from the smaller networks generated based on the same models [7]. Analysis of social networks is relevant to many scientific domains, e.g., data-mining, network sciences, physics, and social sciences [8]. In this section, we briefly investigate social network simulation based on so-called Preferential Attachment [9].

Modeling and implementation of the above-mentioned system is for standard ABS already extensively investigated for both multi-core [10] and distributed [11] architectures in the ABS language. Here we focus on how our proposed notion of streams influences the performance of the system presented in [11]. To adopt data streams, we have modified the communication pattern of the active objects, where instead of one request per message, a batch of requests is sent to an active object via one method invocation and the return values are streamed to the caller via data streaming. The performance gain, discussed below, can be attributed almost entirely to the batching responses instead of sending one packet per return value. Note, such a batching mechanism is integrated naturally in the context of data streams.

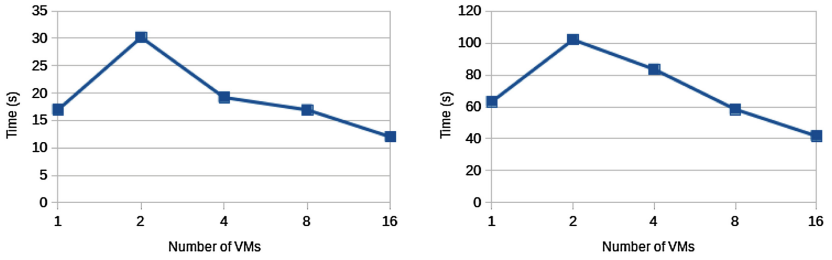
In graph-theoretical terms the problem of Preferential Attachment considers an initial graph of $d+1$ nodes for some small number d and seeks to extend the graph with n nodes, for some $n \ggg d$, by adding nodes one-by-one such that each new node will have degree d. The d target nodes are selected preferentially where the preference is the degree, that is, the higher the degree of a node, the higher the probability to be attached to the new node.

4.1 Experimental Results

The case study on massive social network simulation has been implemented in Cloud ABS [6], which is a source-to-source transcompiler from ABS code down to Cloud Haskell [12] runnable on distributed machines. Beside a higher level of abstraction at the programming level thanks to our proposed feature, the distributed runtime system provides more than 6× speed-up performance compared to the same implementation without using the feature, presented in [11]. The results are illustrated in Fig. 3.

The distribution overhead increases the execution time for two machines, which is compensated by the parallelism achieved through adding more VMs. As shown in Table 1, the memory consumption decreases when adding more VMs, which enables generating extra-large graphs which cannot fit in centralized-memory architectures. We ran the implementation on a distributed cloud

environment kindly provided by the Dutch SURF foundation. The hardware consisted of identical VMs interconnected over a 10Gbps ethernet network; each VM was a single-core Intel Xeon E5-2698, 16GB RAM running Ubuntu 14.04 Server edition. Finally, we provided an online repository[1] containing the ABS code for the case study and instructions for installing the ABS Haskell backend.



(a) (b)

Fig. 3. Performance results of the distributed PA in ABS-Haskell for graphs of $n = 10^7$ nodes with $d =$ (a) 3, (b) 10.

Table 1. Maximum memory residency (in MB) per virtual machine.

	Total number of VMs				
Graph size	1	2	4	8	16
$n = 10^6, d = 3$	306	266	212	155	114
$n = 10^6, d = 10$	899	1028	547	354	221
$n = 10^7, d = 3$	2123	3242	1603	967	621
$n = 10^7, d = 10$	6260	9668	6702	3611	1905

5 Conclusion and Future Work

In the extended ABS, the proposed type `strFut<T>` of asynchronous data streams is similar to that of simple futures in the sense that a value of its type `T` can be passed around. However, shared data streams in general will give rise to race conditions because, by definition, processing an element from the stream implies its removal. Different standard techniques can be used to control race conditions, like ownership types. Alternatively, in future work we will investigate monotonically increasing streams whose generated elements are persistent. This will involve some additional cursor mechanism for local reading devices for different users and requires auxiliary garbage collection techniques.

Work is well under way addressing the type system and operational semantics of the proposed notion as an extension of ABS.

[1] http://github.com/abstools/distributed-PA/streams.

Acknowledgments. This research is partly funded by the EU project FP7-612985 UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations (http://www.upscale-project.eu). This work was carried out on the Dutch national HPC cloud infrastructure, a service provided by the SURF Foundation (http://www.surf.nl). We also thank Erik de Vink for his constructive comments.

References

1. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). doi:10.1007/978-3-642-25271-6_8
2. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007). doi:10.1007/978-3-540-71316-6_22
3. Selivanov, Y.: Asynchronous generators (2016). https://www.python.org/dev/peps/pep-0525/
4. Streams - version 2.5.0 (2017). http://doc.akka.io/docs/akka/2.4/scala/stream/index.html
5. Welch, P., Brown, N.: Communicating Sequential Processes for Java^{tm} (JCSP) (2014). https://www.cs.kent.ac.uk/projects/ofa/jcsp/
6. Bezirgiannis, N., de Boer, F.: ABS: a high-level modeling language for cloud-aware programming. In: Freivalds, R.M., Engels, G., Catania, B. (eds.) SOFSEM 2016. LNCS, vol. 9587, pp. 433–444. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49192-8_35
7. Leskovec, J.: Dynamics of large networks. ProQuest (2008)
8. Bader, D., Madduri, K.: Parallel algorithms for evaluating centrality indices in real-world networks. In: International Conference on Parallel Processing 2006, pp. 539–550. IEEE (2006)
9. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. Science **286**(5439), 509–512 (1999)
10. Azadbakht, K., Bezirgiannis, N., de Boer, F.S., Aliakbary, S.: A high-level and scalable approach for generating scale-free graphs using active objects. In: 31st Annual ACM Symposium on Applied Computing, pp. 1244–1250. ACM (2016)
11. Azadbakht, K., Bezirgiannis, N., de Boer, F.S.: Distributed network generation based on preferential attachment in ABS. In: Steffen, B., Baier, C., Brand, M., Eder, J., Hinchey, M., Margaria, T. (eds.) SOFSEM 2017. LNCS, vol. 10139, pp. 103–115. Springer, Cham (2017). doi:10.1007/978-3-319-51963-0_9
12. Epstein, J., Black, A.P., Peyton-Jones, S.: Towards Haskell in the cloud. ACM SIGPLAN Not. **46**, 118–129 (2011). ACM