Chapter 4

Futures for Streaming Data

4.1 Introduction

Since the rapid growth in big data, data streaming is widely used in many distributed applications, e.g., telecommunications, event-monitoring and detection, and sensor networks. Data streaming is a client/server pattern which, in essence, consists of a continuous generation of data by the server and a sequential and incremental processing of the data by the client. Data streams are naturally processed differently from batch data. Functions cannot operate on data streams as a whole, as the produced data can be unlimited. Hence, new programming abstractions are required for the continuous generation and consumption of data in the streams.

Data streaming is highly relevant in modern distributed systems. Actor-based languages are specifically designed for describing such systems [1]. They provide an event-driven model of concurrency where messages are communicated asynchronously and processed by pattern matching mechanism [7]. Concurrent objects generalize this model to *programming to interface* discipline by modeling messages as asynchronous method invocations. The main contribution of this chapter is to integrate data streaming mechanism with concurrent object systems.

In this chapter, we extend the ABS language in order to support the streaming of data between a server and its clients. We introduce "future-based data streams" which integrates futures and data streams, and which specifies the return type of so-called *streaming* methods. Upon invocation of such a method a new future is created which holds a reference to the generated stream of data. Data items are added to the stream by the execution of a *yield* statement. Such a statement takes as parameter an expression the value of which is added to the stream, *without* terminating the execution of the method. The *return* statement terminates the execution of a streaming method, and is used to signal the end of data streaming. Even though no new data is produced, the existing data values in the stream buffer can be retrieved by the consumers.

The values generated by the server (the streaming method) can be obtained incrementally and sequentially by a client by querying the future corresponding to this method invocation. By the nature of data streaming, it is natural to restrict the streaming to the asynchronous method calls. Therefore there is no support for synchronous invocation of streaming methods.

In this chapter, we introduce two different implementations of streams tailored to different forms of parallel processing of data streams. Obtaining data from a *destructive* stream involves the removal of the data, whereas in a *non-destructive* stream the data persists. Which of the implementation is used is determined by the caller of the streaming method (the *creator* of the stream) which is not necessarily the consumer of the data stream. The creator can then provide the consumers with a reference to the stream. Both the streaming method (producer) and the consumers which hold a reference to the data stream are *not* exposed to the underlying implementation of the stream, i.e., these different implementations are not represented by different *types* of data streams. This allows for a separation of concerns between the generation and processing of data streams, on the one hand, and their orchestration, on the other hand. This also enables reusability of the implementation of producers and consumers for both consumption approaches.

A preliminary discussion of the overall idea underlying this chapter is given in [11]. As an extension, in this chapter we introduce the different implementations of data streams, an operational semantics for both implementations of streams, a new type system which formalizes the integration of futures and data streams, and a proof of type-safety. Further, we show how the basic mechanism in ABS of cooperative scheduling of asynchronously generated method invocations itself can be used to implement data streams and the cooperative scheduling of streaming methods.

As a proof of concept, exploiting a prototype implementation for supporting future-based data streams on top of ABS, we present the usage of the above-mentioned feature in the implementation of a distributed application for the generation of distributed PA (chapter 2 and 3). 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 [20] for the case study on future-based data streams reported in this chapter.

The overall contribution of this chapter is a formal model of streaming data in the ABS language, which fully complies and generalizes the asynchronous model of computation underlying the ABS language. Since ABS is defined in terms of a formal operational semantics which supports a variety of formal analysis techniques (e.g., deadlock detection [39] and [14]), we thus obtain a general formal framework for the modeling and analysis of different programming techniques for processing data streams, e.g., *map-reduce* and *publish-subscribe* [34]. To the best of our knowledge, our work provides a first formal type system and operational semantics for a general notion of streaming data in a high-level actor-based programming language.

Plan of the chapter This chapter is organized as follows: the notion of a future-based data stream is specified as an extension of ABS in Section 4.2. In section 4.3, it is shown that the well-typedness of a program in the extended ABS is preserved. Section 4.4 discusses the usage of streams in a distributed setting. In section 4.5, an implementation of data streams is given as an API written in ABS. In Section 4.6, a case study on social network simulation is discussed, which uses the proposed notion of streams. Related works are discussed in section 4.7. Finally we conclude in section 4.8.

4.2 Future-Based Data Streams

In this section, we define future-based data streaming in the context of the ABS language. A *streaming* method is statically typed, namely, the return type of the method is followed by the keyword **`stream`**, specifying that the method streams values of that type. As mentioned before, ABS features a programming to interfaces discipline. Therefore the caller can asynchronously call a streaming method, provided that the interface of the callee includes the method definition.

Data streaming is defined as a stream of return values from a callee to the data consumers of the stream in an asynchronous fashion. An invocation of a streaming method creates a stream. The callee first create an empty stream, and then produces and stores data to the stream buffer via the **`yield`** statement. The caller assigns the invocation to a variable of type **`Stream`**`<T>` for the return type `T` **`stream`** of the callee. The stream variable can be passed around. Therefore different variables in multiple processes (a process is the execution of an asynchronous method call) may refer to the same stream and retrieve data from it.

We distinguish between two different kinds of streams: *destructive* and *non-destructive* streams. The kind of stream is determined by the caller upon the invocation of the streaming method. In destructive streams, values are retrieved from a FIFO queue which stores the data produced but not yet consumed. Querying availability of data values in an empty queue gives rise to a cooperative release of control (further discussed below). Also an attempt to take a value from a stream where the callee is terminated (and thus no further data streaming will take place) gives rise to the execution of a block of statements specified by programmer for this reason, thus avoiding the generation of a corresponding error (see below). Parallel processes which have access to the same destructive stream compete for the different data items produced. Consequently, the parallel processing of destructive data streams gives rise to *race conditions*, in the sense that different order of requests to read from a stream may correspondingly give rise to different data values. Note that at most one process can destructively read a specific data value. On the other hand, a non-destructive stream allows complete *sharing* of all the data produced which are only *read* to be processed. As described in more detail below, non-destructive streams maintain access by means of cursors at different positions of the buffer which allows for its asynchronous parallel processing.

Abstracting from the typing information, to be discussed in more detail below, the syntax of our proposed extension of ABS, i.e., that of future-based data streams, is specified in Figure 4.1, where e denotes an expression (i.e., a variable name, etc), $\overline{e}$ denotes a sequence of expressions, x is a variable name, m is a method name, and s denotes a sequence of statements.

$$\begin{array}{rl} s ::= & s; s \mid x = [\textbf{nd}]\ e!m(\overline{e}) \mid \textbf{yield}\ e \mid \textbf{return} \mid \textbf{suspend} \mid \\ & \textbf{await}\ e?\ \textbf{finished}\ \{s\} \mid x = e.\textbf{get}\ \textbf{finished}\ \{s\} \end{array}$$

Figure 4.1: Syntax

In the asynchronous invocation $x = [\textbf{nd}]\ e!m(\overline{e})$ of a streaming method, the optional keyword **nd** indicates the creation of a new non-destructive stream.

Execution of the **yield** statement, which can only be used in the body of a *streaming* method, consist of queuing the value of the specified expression.

Execution of the **return** statement by a streaming method indicates termination of the data generation which is signaled to the consumers of the stream by queuing the special value η.

The active process unconditionally releases control by **suspend**. The object is then idle and can activate a suspended process. The **await-finished** statement allows to check the buffer of the stream denoted by the expression e in the following manner: if there is at least one proper value, different from the signal η, in the buffer, the statement is skipped. In case the buffer is empty, the current process *suspends* such that the object can activate another process. The statement s is executed in case the buffer only contains the signal η.

The **get-finished** statement allows to actually retrieve (in case of a destructive stream) or read (in case of a non-destructive stream) a next data value. It however *blocks* the whole object when the buffer is empty. As above, statement s is executed when the buffer only contains the signal η.

In **await-finished** and **get-finished**, the keyword **finished** and its following block can be omitted if the block is empty.

We next illustrate the difference in the behaviour of destructive and non-destructive access to a stream by the following simple toy example. Consider the streaming method `m()`:

```
Int stream m() {
  yield 1; yield 2; return;
}
```

This method adds 1 and 2, followed by a termination token to the resulting stream buffer.

The following snippet asynchronously calls the above method definition `m()` on some object `o` which gives rise to two references `r1` and `r2` to the resulting destructive stream.

```
Stream<Int> r1 = o!m();
Stream<Int> r2 = r1;
```

The following code uses the above references, with the assumption that it is the only process that consumes data items of the stream:

```
    Int x, y, z;
(1) Int x = r1.get finished {x = -1};
(2) Int y = r2.get finished {y = -1};
(3) Int z = r1.get finished {z = -1};
```

Once the process corresponding to the method call `m()` on (possibly remote) object `o` is executed and the results are provided to the stream, the values 1, 2, and -1 are assigned to `x, y` and `z`, respectively. These values are consumed from the stream and assigned to the variables incrementally as soon as they are provided by `m()`. In the above code, the object possibly blocks on any of the three statements, if a value (whether an integer or the terminating token) is not yielded to the stream yet. The statement `(1)` destructively reads 1 from the stream via `r1` and assigns it to `x`. The statement `(2)` destructively reads 2 from the same stream via the other reference `r2` and assigns it to `y`. However, the statement `(3)` runs the **`finished`** statement which assigns -1 to `z`, since it reads the terminating token (i.e., the stream is already terminated). Any further *get* operations on *every* variable referring to the stream also read the terminating token.

To show how a non-destructive stream works in the same setting, suppose we use the following references `r1` and `r2` in the above code (note that the keyword **`nd`** denotes that the resulting stream is non-destructive).

```
Stream<Int> r1 = nd o!m();
Stream<Int> r2 = r1;
```

With the same incremental production of values and blocking mechanism, in this setting the values 1, 1, and 2 are assigned to `x, y` and `z`, respectively. The statement `(1)` non-destructively reads 1 from the stream via `r1` and assigns it to `x`. The statement `(2)` non-destructively reads 1 from the same stream via the other reference `r2` (with its own cursor to the stream) and assigns it to `y`. Finally, the statement `(3)` assigns 2 to `z`, since the cursor of `r1` is already moved forward by statement `(1)`. Note that any number of further *get* operations on `r1` will read the terminating token. It is important to observe the role of cursors per each stream variable that gives rise to such behaviour.

Note that the assignment of a non-destructive reference (`r2 = r1`) is different from the standard ABS assignment in the sense that, in addition to the assignment of the reference to stream, it also assigns the cursor. Based on this design, the copying is required as each stream variable represents a new access to the stream to *all* data values from the position its cursor denotes.

4.2.1 Design Decisions

Integration of streams with ABS, where we enjoy the advantages of both, roots in the ever-growing application of data streaming in different domains. The consumption approaches of the stream (i.e., destructive or non-destructive) are not fixed in the streams in form of different data types. Instead, the creator of the stream determines the consumption approach of the stream instance, in order to maintain generality. Note that the creator of the stream is not necessarily the consumer of the stream, and by design, it can be considered as part of the producer process (e.g., using *factory method* design pattern) that forces one of the above consumption approaches to the consumers.

We support both destructive and non-destructive data streams, as they can be naturally used to implement, respectively, *one-of-n* semantics (only one consumer reads a given data as in, e.g., *data parallelism* model), and *one-to-n* message delivery (a given data can be read by all such consumers as in, e.g., one-to-many *trainer and learners* and *publish/subscribe* model). Also integration of data streaming and cooperative scheduling enables enhancing concurrency and parallelism on the consumer side.

Note that the above two approaches allow for designing a third hybrid consumption approach where, at the intra-object level, every access to the stream buffer is via an object field (shared variable), and at the inter-object level, the cursor is copied (i.e., via passing parameters in method invocations).

4.2.2 Example of Destructive Streams

The code example in Figure 4.2 illustrates the use of ABS destructive data streams in modeling a parallel *map-reduce* processing of a data stream. The mapping step maps each streamed data value of type `T` to a data value of type `Int`, and the reduction step calculates the average of those integers.

An ABS program is a set of interface and class definitions, followed by the main block of the program, which is an anonymous block at the end of the program. The main block is the initial run-time process (similar to `public static void main` in java). Each class implements at least one interface. The type of a reference variable to an object can only be an interface, and the object must be an instance of a class that implements the interface. Every object instance is an active object, namely, it features a dedicated thread of control, and can have (at most) one active process among its processes. Each process of an object is initiated by an asynchronous call of a method of the object.

```
interface IMapper<T> {
 Int stream map(Stream<T> s);
}

interface IReducer<T> {
 Pair<Int, Int> reduce(Stream<T> s);
}
interface IPar<T> {
 Int start(Stream<T> s, Int num);
}
interface IProd<T> {
  T stream streamer();
}
class Mapper()
implements IMapper<T> {
 Int stream map(Stream<T> s) {
   Bool last = False;
   while(last==False){
     T v = s.get finished {last=True};
     if (last == False) {
       yield v.value();
     }
   }
   return;
 }
}
class Reducer()
implements IReducer<T> {
 Pair<Int, Int> reduce(Stream<Int> s)
 {
   Bool last = False;
   Int count = 0;
   while(last==False){
     Int v = s.get
       finished {last=True};
     if (last == False){
       count = count + 1;
       sum = sum + v;
     }
   }
   return Pair(sum, count);
 }
}
```

```
class Producer implements IProd<T>
{
  T stream streamer() {
    // yields a seq of data of type T
    return;
  }
}
class Par implements IPar<T> {
 Int start(Stream<T> s, Int num) {
   Int m = 1;
   Int sum = 0, avg = 0, count = 0;
   List<Fut<Pair<Int, Int>>> l = Nil;
   while(m<=num) {
     IMapper<T> p = new Mapper();
     Stream<Int> s2 = p!map(s);
     IReducer<T> q = new Reducer();
     l = Cons(q ! reduce(s2), l);
     m=m+1;
   }
   while (l != Nil) {
    Pair<Int,Int> pair = head(l).get;
    case (pair) {
      Pair(a, b) => {
          sum = sum + a;
          count = count + b;
      }
    }
    l = tail(l);
   }
   if (count > 0) return sum / count;
   else return 0;
 }
}
{// Main block
 IProd<T> producer = new Producer();
 Stream<T> s = producer ! streamer();
 IPar<T> par = new Par();
 Int average = par.start(s, 4);
}
```

Figure 4.2: Parallel data processing based on Map-Reduce data model

The program is composed of four interfaces: `IProd` types a class with a streaming method to stream the data values of type `T` to be processed. The interface `IPar` types a class for spawning multiple chains of active objects for map-reduce processing. Each chain is a pipeline processing of the data values retrieved from the stream which is shared among the chains. The interfaces `IMapper` and `IReducer` type the objects that form a pipeline chain. These interfaces are implemented by four classes `Producer`, `Par`, `Mapper` and `Reducer`, respectively. The above definitions are followed by the main block of the program. As shown in the main block, the general idea is that the data values of the stream `s` will be processed in parallel by `num` computationally identical pipelines, and the aggregated result, which is the average of those values, is returned as the final result. Runtime control and data flow of the example are also illustrated in Figure 4.3, where each thread represents a process created by an asynchronous method call.

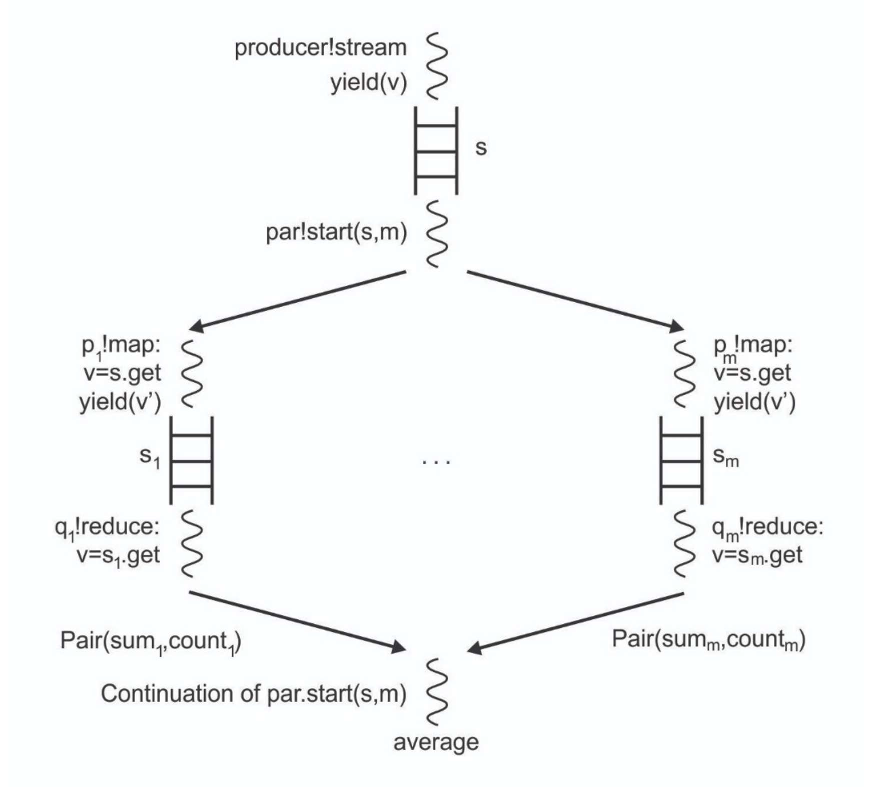

Figure 4.3: Control and data flow

The asynchronous invocation of method `streamer` on `producer` in the main block returns a reference `s` to a stream. The method `start` provided by `IPar<T>` enables parallel processing of the stream by creating multiple chains (`num`) of two active objects of type `IMapper<T>` and `IReducer<T>`, where the former retrieves values from `s`, and yields a mapped value of type integer to an intermediate stream `s2`, and the latter consumes those integers from `s2` and reduces them to a pair which is the `sum` and the `count` of those integers processed by one chain. The futures of the pairs resulting from calling `reduce` in different chains are stored in list `l`. The elements of the list are then used as a synchronization means, namely, awaiting until each process resolves the corresponding future by providing the return value. Finally the average is calculated by the `start` from the reduced pairs.

Similar to parallel map-reduce transformations on streams in languages like Scala, the following pseudo-code can be used as a simplified abstract replacement for the code in 4.2:

```
s.par(num).map(_.value).average();
```

where a sequence of *transformation* methods (e.g., *map* and *filter*) followed by a *reduce* method (e.g., *count* and *average*) can be executed in parallel by *num* threads (modeled by active objects) on stream `s`.

Note that our implementation utilizes two ways of parallelism: 1) *horizontal parallelism*, which is achieved by creating multiple chains of active objects, e.g., p_i and q_i and intermediate streams s_i in Figure 4.3, and 2) *vertical parallelism*, which is achieved by pipeline processing, e.g., the process *map* in p_i that yields values to s_i runs in parallel with *reduce* in q_i that consumes the values immediately upon their availability.

4.2.3 Example of Non-Destructive Streams

In the example specified in Fig. 4.4, we represent a basic means of *publish/subscribe* communication via non-destructive streams in a social network such as Twitter. An object of class `Member` denotes a member in the network that can follow and be followed by multiple members. The main idea is to implement each member object such that: 1) it can follow multiple members by reading their stream of posts 2) its stream of posts can be read by multiple members that follow the member 3) it can post new items to its stream. The object naturally needs to interleave these tasks. To this aim, each member is modeled as an actor with a process to post new items to its stream (`share`), a set of processes one per each member it follows, in order to read their streams (`follow`), and a set of processes from other members that request to follow the member (`request`). These processes can be interleaved by the ABS *cooperative scheduling*. The active process can cooperatively release control conditionally, e.g., the **`await`** statement in `follow` which checks whether there is no new post to be read from a specific member, or unconditionally, e.g., the **`suspend`** in `share` after posting a new item gives rise to unconditional release of control. In both cases, other processes of the member object can be activated.

The method `follow` sends a request to a member denoted by the argument p. The data (i.e., posts) can be retrieved from the resulting stream r of the member `p` by the current member. In other words, the current member object *follows* object p. Further, a followed member returns a reference to the same data stream for all the followers, denoted by r in the the class `Member`. Each follower uses its corresponding cursor to read from the stream belonging to the followed member. Note the difference between the return types of `share` and `request`. The former is a streaming method that creates and populates a stream, and can *only* be called asynchronously with the return type **`Stream`**`<Post>`, whereas the latter is a non-streaming method that returns a reference to an existing stream, and returns **`Stream`**`<Post>` or **`Fut`**`<`**`Stream`**`<Post>>`, respectively, depending on being called synchronously or asynchronously.

```
interface IMem {
  Unit run();
  Unit follow(IMem p);
  Stream<Post> request();
}

class Member implements IMem
{
  Stream<Post> r;
  // r is a stream of
  // posts for followers

  Unit run() {
    if (r == null)
      r = nd this!share();
  }

  Post stream share() {
    Post post;
    while(True) {
      // Next post is ready
      yield post;
      suspend;
    }
    return;
  }
```

```
  Unit follow(IMem p) {
    Fut<Stream<Post>> f =
      p ! request();
    await f?;
    Stream<Post> r = f.get;
    Bool last = False;
    while(last = False) {
      await r? finished
      {
        //probably p left!
        last = true;
      }
      Post post = r.get;
      // consume post
    }
  }

  Stream<Post> request() {
    // accept as a follower
    return r;
  }
}
```

Figure 4.4: Parallel data processing based on publish/subscribe pattern

By **`await`** statement, a *follow* process queries the availability of the next post that is *new* from the perspective of the non-destructive stream variable `r`, denoted by the variable cursor. If the new post is available it is retrieved and consumed. Otherwise the process is suspended so that another enabled process is activated. As such, the member receives posts from all the members it follows, processes the follow *requests* of other members, and posts new data. The stream corresponding to a followed member can signal the termination. In such case, the `follow` process in the follower object which corresponds to a followed member terminates after retrieving the remaining posts, as the *finished* block of the await statement falsifies the loop condition. The process that instantiates a new member (not mentioned here) also initiates the member by calling the `run` which itself calls the `share` method which returns a new stream and continuously adds new posts to it.

4.2.4 Type System

The ABS type system is presented in [48]. An extension of the type system is specified below using the same notation, which types the streams and the statements that use them (Figure 4.5). A *typing context* Γ is a mapping from names to types, where the names can be variables, constants and method names. A *type lookup* is denoted by $\Gamma(x)$, which returns the type of the name x. By $\Gamma[x \mapsto T]$ we denote the update of Γ such that the type of x is set to T. Then $\Gamma[x \mapsto T](x) = T$ and $\Gamma[x \mapsto T](y) = \Gamma(y)$ if $x \neq y$. An over-lined $\overline{e}$ denotes a sequence of syntactic entities e.

The basic idea underlying the typing rules regarding streams in Figure 4.5 is that the type $\mathsf{stream}\langle T\rangle$ of streams of data items of type T itself can*not* be defined as a *subtype* of $\mathsf{fut}\langle T\rangle$, since for a future variable x, a query $x?$ gives rise to a Boolean guard whereas for a stream variable x, the query $x?$ is not a Boolean guard because it not only checks whether the stream is empty or not but also whether it has terminated. On the other hand, the type $\mathsf{fut}\langle T\rangle$ of futures that refer to return values of type T itself can be defined as a sub-type of $\mathsf{stream}\langle T\rangle$ (as specified by the rule T-FutureStream), where the stream buffer is either empty (denoted by a sentinel $\perp$) or contains an *infinite* sequence of the particular return value. For such streams the **`finished`** statement never executes, as there is no termination token. Note also that for such infinite streams there is no difference between destructive or non-destructive reads.

We proceed with a brief explanation of the typing rules. A streaming method is well-typed by T-StreamMethod, if its body s is well-typed in the typing context extended by the parameters, local variables, and the return stream. The *destiny* variable in ABS is a local variable which holds a reference to the return stream (or future). A regular (non-streaming) method is similarly well-typed by T-Method in core ABS.

By T-AsyncCall, an asynchronous method call to a non-streaming method has type $\mathsf{fut}\langle T\rangle$, if its corresponding synchronous call has type T. Whereas by T-AsyncStream the type of an asynchronous call of a streaming method is of type $\mathsf{stream}\langle T\rangle$, if the interface T' of the callee includes the streaming method. As in ABS, by T-SyncCall, a call to a method m has type T if its actual parameters have types $\overline{T}$ and the signature $\overline{T} \rightarrow T$ matches a signature for m in the known interface of the callee (given by an auxiliary function *match*). The rule does not allow synchronous call on a streaming method (note the difference between how the function *match* is used in T-AsyncStream and T-SyncCall).

The **yield** statement is well-typed in a streaming method by T-Yield, if the type of e is T and the enclosing method is a streaming method of type T. **return** statement without parameter is only used in a streaming method to signal the termination of streaming and the method, and is well-typed by T-ReturnStream.

The T-Return forces that the expression e of the **return** statement in a non-streaming method is of type T, the return type of the enclosing method.

The **await-finished** is well-typed by T-AwaitStream, if a stream of some type T is awaited and if the statement s is also well-typed. It is not difficult to see how the statement **get-finished** is well-typed by T-GetStream.

$$\text{(T-FutureStream)}\quad \frac{\Gamma \vdash e : \mathsf{fut}\langle T\rangle \qquad T \preccurlyeq T'}{\Gamma \vdash e : \mathsf{stream}\langle T'\rangle}$$

$$\text{(T-Method)}\quad \frac{\begin{array}{c}\Gamma' = \Gamma[\overline{x} \mapsto \overline{T}, \overline{x'} \mapsto \overline{T'}]\\ \Gamma'[\text{destiny} \mapsto \mathtt{fut}\langle T''\rangle] \vdash s\end{array}}{\Gamma \vdash T''\ m\ (\overline{T}\ \overline{x})\{\overline{T'}\ \overline{x'}; s\}}$$

$$\text{(T-StreamMethod)}\quad \frac{\begin{array}{c}\Gamma' = \Gamma[\overline{x} \mapsto \overline{T}, \overline{x'} \mapsto \overline{T'}]\\ \Gamma'[\text{destiny} \mapsto \mathsf{stream}\langle T''\rangle] \vdash s\end{array}}{\Gamma \vdash T''\ \mathsf{stream}\ m(\overline{T}\ \overline{x})\ \{\overline{T'}\ \overline{x'}; s\}}$$

$$\text{(T-ReturnStream)}\quad \frac{\Gamma(\text{destiny}) = \mathsf{stream}\langle T\rangle}{\Gamma \vdash \mathtt{return}}$$

$$\text{(T-AsyncCall)}\quad \frac{\Gamma \vdash e.m(\overline{e}) : T}{\Gamma \vdash e!m(\overline{e}) : \mathsf{fut}\langle T\rangle}$$

$$\text{(T-SyncCall)}\quad \frac{\begin{array}{c}\Gamma \vdash e : T' \qquad \Gamma \vdash \overline{e} : \overline{T}\\ \text{match}(m, \overline{T} \to T, T')\end{array}}{\Gamma \vdash e.m(\overline{e}) : T}$$

$$\text{(T-Return)}\quad \frac{\begin{array}{c}\Gamma \vdash e : T\\ \Gamma(\text{destiny}) = \mathsf{fut}\langle T\rangle\end{array}}{\Gamma \vdash \mathtt{return}\ e}$$

$$\text{(T-AsyncStream)}\quad \frac{\begin{array}{c}\Gamma \vdash e : T' \qquad \Gamma \vdash \overline{e} : \overline{T}\\ \text{match}(m, \overline{T} \to T\ \ \mathsf{stream}, T')\end{array}}{\Gamma \vdash [\mathbf{nd}]\ \ e!m(\overline{e}) : \mathsf{stream}\langle T\rangle}$$

$$\text{(T-Yield)}\quad \frac{\Gamma \vdash e : T \qquad \Gamma(\mathit{destiny}) = \mathsf{stream}\langle T\rangle}{\Gamma \vdash \mathtt{yield}\ e}$$

$$\text{(T-AwaitStream)}\quad \frac{\Gamma \vdash e : \mathsf{stream}\langle T\rangle \qquad \Gamma \vdash s}{\Gamma \vdash \mathtt{await}\ \ e?\ \ \mathtt{finished}\ \{s\}}$$

$$\text{(T-GetStream)}\quad \frac{\Gamma \vdash e : \mathsf{stream}\langle T\rangle \qquad \Gamma \vdash s \qquad \Gamma \vdash x : T}{\Gamma \vdash x = e.\mathtt{get}\ \ \mathtt{finished}\ \{s\}}$$

Figure 4.5: Type system

4.2.5 Operational Semantics

The operational semantics of the proposed extension is presented below as a transition system in SOS style [61]. First we extend the ABS run-time configuration and then present those rules in the transition system that involve destructive and non-destructive streams.

Runtime Configuration

The runtime syntax of ABS is extended by the notion of stream is illustrated in Figure 4.6. Configurations *cn* consist of objects (*object*), invocation messages (*invoc*), futures (*fut*), and data streams (*stream*). The commutative and associative composition operator on configurations is denoted by whitespace. The empty configuration is denoted by ϵ.

$$
\begin{array}{rcl@{\qquad}rcl}
cn & ::= & \epsilon \mid fut \mid stream \mid & & & \\
 & & object \mid invoc \mid cn\ cn & & & \\
object & ::= & ob(o, a, p, q) & p & ::= & process \mid \texttt{\textbf{idle}} \\
fut & ::= & fut(f, value) & a & ::= & T\ x\ v \mid a, a \\
stream & ::= & stream(f, u) & value & ::= & v \mid \bot \\
process & ::= & \{a \mid s\} \mid \texttt{\textbf{error}} & u & ::= & u.u \mid v \mid \eta \mid \bot \\
q & ::= & \epsilon \mid process \mid q\ q & v & ::= & o \mid f \mid t \mid (f, n) \\
invoc & ::= & invoc(o, f, m, \overline{v}) & & &
\end{array}
$$

Figure 4.6: Runtime configuration

The term $ob(o, a, p, q)$ represents an object where o is the object identifier, a assigns values to the object's fields, p is an active process (or idle), and q represents a set of suspended processes.

The term $invoc(o, f, m, \overline{v})$ represents an invocation message, where o is the callee object, f is the identifier of a rendezvous for the return value(s) of the method invocation which can be a stream or a future, depending on the invoked method being a *streaming* method or not, m is the name of the invoked method, and $\overline{v}$ are its arguments.

A process $\{a \mid s\}$ consists of an assignment a of values to the local variables, and a statement s. A process results from the activation of a method invocation in a callee with actual parameters, and an associated future or stream. An **`error`** is a process where the binding of such method invocation does not succeed.

A future is represented by $fut(f, value)$, where f is the future identifier and *value* denotes its current value which can either be the actual value returned or $\bot$ which denotes the absence of a return value.

Both destructive and non-destructive streams are semantically represented by $stream(f, u)$, where f is the stream identifier, and u denotes its buffer. Nevertheless, the stream variables referring to destructive streams just hold the stream id, whereas the value of a stream variable referring to a non-destructive stream is a pair (f, n), where n denotes the associated position in the stream.

The buffer u is a FIFO queue, which contains a sequence of values v, and a special symbol, either $\bot$ which is a sentinel denoting end of buffer, or η which denotes termination of streaming. The $\bot$ is replaced by η after adding the last valid value to the queue when the streaming method terminates. The $u = v.u'$ denotes the head v of the queue u, and its tail u'. In $u' = u.v$, enqueuing the value v to the end of the queue u forms the updated queue u'. The auxiliary function $\mathrm{elem}(u, n)$ returns the content at the position n of the sequence u starting from 0.

A value v can be an object identifier, a future or stream identifier, a term t which is a value of a primitive type, or a pair (f, n) which is a value of a variable referring to a non-destructive stream.

Note that all the identifiers in a configuration are unique and terminal: o is used for object, and f both for future and stream identifiers.

The rules of Figure 4.8 and 4.9 operate on the elementary configurations. To have the rules to apply to full configurations, we need the following rule as well:

$$\frac{cn' \to cn''}{cn\ cn' \to cn\ cn''}$$

We also use the reduction system proposed in the ABS formal model to evaluate expressions, e.g., $f = [\![e]\!]^{cn}_{a \circ l}$ in the active process of $ob(o, a, \{l|s\}, q)$ holds if the expression e evaluates to the stream identifier f, in an assignment composed of a and l, where the configuration $cn\ ob(o, a, \{l|s\}, q)$ is given, and cn contains $stream(f, u)$. By definition, $a \circ l(x) = l(x)$ if $x \in dom(l)$ or $a \circ l(x) = a(x)$ otherwise.

The following rule Async-Call represents asynchronous method invocation in core ABS extended with a check that it is not a streaming method:

$$\text{(Async-Call)}\quad \frac{o' = [\![e]\!]_{(a \circ l)} \quad \overline{v} = [\![\overline{e}]\!]_{(a \circ l)} \quad \mathrm{fresh}(f) \quad \neg\mathrm{streamer}(o'.m(\overline{v}))}{\begin{array}{c} ob(o, a, \{l|x = e!m(\overline{e}); s\}, q) \\ \to ob(o, a, \{l|x = f; s\}, q) \quad invoc(o', f, m, \overline{v}) \quad fut(f, \bot) \end{array}}$$

where it sends an invocation message to object o' with the method name m, the future f and the actual parameters $\overline{v}$. The return value of f is undefined (i.e., $\bot$). Note that, based on Figure 4.6, the definition of v also includes the values f and (f, n) for destructive and non-destructive streams in the extended semantics. Therefore streams can be passed as actual parameters and assigned to formal parameters.

Also for the chapter to be self-contained, the following rules from the core ABS [48] are mentioned. Rules Assign-Local and Assign-Field assign value of expression e to the variable in, respectively, environment l for local variables or environment a for fields of object a. Rule Suspend suspends the active process unconditionally. Rule Activate activates a process p that is *ready* to execute for the idle object o from the set q of its suspended processes.

(Assign-Local)
$$\frac{x \in \mathrm{dom}(l) \quad v = [\![e]\!]_{(a\circ l)}}{ob(o, a, \{l | x = e; s\}, q) \\ \to ob(o, a, \{l[x \mapsto v] | s\}, q)}$$

(Assign-Field)
$$\frac{x \in \mathrm{dom}(a) \quad v = [\![e]\!]_{(a\circ l)}}{ob(o, a, \{l | x = e; s\}, q) \\ \to ob(o, a[x \mapsto v], \{l | s\}, q)}$$

(Suspend)
$$ob(o, a, \{l | \textbf{suspend}; s\}, q) \\ \to ob(o, a, \mathrm{idle}, q \cup \{l | s\})$$

(Activate)
$$\frac{p = \mathrm{select}(q, a)}{ob(o, a, \mathrm{idle}, q) \to ob(o, a, p, q \backslash p)}$$

The auxiliary method select(q, a, l) selects the ready process by ensuring the process will not be immediately re-suspended based on the states of a and l. Note that we abstract from the notion of the ABS concurrent object group *cog* in the rule. Also $dom(a)$ denotes the set of variables in the environment a.

In the rest of this section, we present the semantic rules of the extended ABS, where a data stream is involved. Given in Figure 4.7, the rules for the callee side, which only write to the stream, are independent from how the stream is read (i.e., destructively or non-destructively). In the rule Yield, the active process, which is a streaming method, enqueues the value v to the buffer of the stream f, followed by the sentinel $\perp$. The rule ReturnStream enqueues the value η to the buffer of the stream f, which is a token denoting termination of streaming values in the buffer.

(Yield)
$$\frac{v = [\![e]\!]^{cn}_{aol} \quad l(\mathrm{destiny}) = f}{ob(o, a, \{l | \textbf{yield}\ e; s\}, q) \quad stream(f, u.\perp) \\ \to ob(o, a, \{l | s\}, q) \quad stream(f, u.v.\perp)}$$

(ReturnStream)
$$\frac{l(\mathrm{destiny}) = f}{ob(o, a, \{l | \textbf{return}; s\}, q) \quad stream(f, u.\perp) \\ \to ob(o, a, \textbf{idle}, q) \quad stream(f, u.\eta)}$$

Figure 4.7: Operational semantics of streams on the callee side

In the following, the rules for destructive and non-destructive access to the data stream are given. Note that the D and ND are prefixed to the rule names, which stand for the destructive and non-destructive streams, respectively.

Semantics of Destructive Streams

In the rule D-AsyncCall, the object o calls asynchronously a streaming method m with arguments $\overline{v}$ on object o'. The return stream is destructive with the fresh identifier f as the access mode to the return stream of a streaming method is destructive by default. We also use two auxiliary functions in this rule as follows: the function *streamer*$(o.m(\overline{v}))$ checks if the method $m(\overline{v})$ of the object o is a streaming method. The function *fresh*(f) guarantees that the newly introduced name f is not already used in the system.

$$\text{(D-AsyncCall)}\quad \frac{o' = [\![e]\!]^{cn}_{aol} \quad \overline{v} = [\![\overline{e}]\!]^{cn}_{aol} \quad \text{fresh}(f) \quad \text{streamer}(o'.m(\overline{v}))}{\begin{array}{c} ob(o, a, \{l|x = e!m(\overline{e}); s\}, q) \rightarrow \\ ob(o, a, \{l|x = f; s\}, q) \quad invoc(o', f, m, \overline{v}) \quad stream(f, \bot) \end{array}}$$

$$\text{(D-AwaitTrue)}\quad \frac{f = [\![e]\!]^{cn}_{aol}}{\begin{array}{c} ob(o, a, \{l|\textbf{await}\ \ e?\ \textbf{finished}\ \{s_1\}; s_2\}, q) \quad stream(f, v.u) \rightarrow \\ ob(o, a, \{l|s_2\}, q) \quad stream(f, v.u) \end{array}}$$

$$\text{(D-AwaitFalse)}\quad \frac{f = [\![e]\!]^{cn}_{aol}}{\begin{array}{c} ob(o, a, \{l|\textbf{await}\ \ e?\ \ \textbf{finished}\ \{s_1\}; s_2\}, q) \quad stream(f, \bot) \rightarrow \\ ob(o, a, \{l|\textbf{suspend}; \textbf{await}\ \ e?\ \textbf{finished}\ \{s_1\}; s_2\}, q) \quad stream(f, \bot) \end{array}}$$

$$\text{(D-AwaitTerminate)}\quad \frac{f = [\![e]\!]^{cn}_{aol}}{\begin{array}{c} ob(o, a, \{l|\textbf{await}\ \ e?\ \ \textbf{finished}\ \{s_1\}; s_2\}, q) \quad stream(f, \eta) \rightarrow \\ ob(o, a, \{l|s_1; s_2\}, q) \quad stream(f, \eta) \end{array}}$$

$$\text{(D-GetTrue)}\quad \frac{f = [\![e]\!]^{cn}_{aol}}{\begin{array}{c} ob(o, a, \{l|x = e.\textbf{get}\ \ \textbf{finished}\ \{s_1\}; s_2\}, q) \quad stream(f, v.u) \rightarrow \\ ob(o, a, \{l|x = v; s_2\}, q) \quad stream(f, u) \end{array}}$$

$$\text{(D-GetTerminate)}\quad \frac{f = [\![e]\!]^{cn}_{aol}}{\begin{array}{c} ob(o, a, \{l|x = e.\textbf{get}\ \ \textbf{finished}\ \{s_1\}; s_2\}, q) \quad stream(f, \eta) \rightarrow \\ ob(o, a, \{l|s_1; s_2\}, q) \quad stream(f, \eta) \end{array}}$$

Figure 4.8: Operational semantics of destructive streams

The await statement in rule D-AwaitTrue is skipped as there exists a data value v in the buffer. By rule D-AwaitFalse, the process querying the empty (but not-yet-terminated) stream f will be suspended. To this aim, the statement **suspend** for unconditional suspension is added to the beginning of the sequence of the statements of the process. According to the standard ABS, the **suspend** then suspends the active process, namely, it adds the process to q, where the active object is *idle* and ready to activate a suspended process from q. In rule D-AwaitTerminate, the *finished* block s_1 of the statement is selected for execution, since the streaming is terminated, i.e., the head of the buffer of the stream f is equal to the terminating token η.

The rule D-GetTrue assigns the value v from the head of the stream buffer to the variable x destructively, i.e., v is removed from the buffer. By D-GetTerminate, the **`finished`** block s_1 of the statement is executed followed by s_2, as the terminating token is observed at the head of the buffer. Note that the state of x remains the same. There is no rule for the **`get-finished`** statement when the buffer is empty which implies that the active process (and the object) is *blocked* until the buffer contains an element.

Semantics of Non-Destructive Streams

The operational semantics of ABS for those rules that involve non-destructive future-based streams is given in Fig. 4.9. In ND-AsyncCall, an asynchronous call to a streaming method m in o' is given with the actual parameters $\overline{v}$, that results in a reference to a non-destructive stream. The keyword **`nd`** denotes the non-destructive access to the resulting stream. Therefore, the return reference to the newly created stream with identifier f is a pair of f and a cursor which is initialized to 0, denoting the first position in the buffer which is initially $\perp$.

Note that by the before-mentioned Assign-Local and Assign-Field, an assignment $x = y$, where the variable y referring to a non-destructive stream f (i.e., $(f, n) = [\![y]\!]^{cn}_{aol}$), the variable x also refers to the same stream and the cursor of x is initialized to the same position in the buffer as the one of y (i.e., $(f, n) = [\![x]\!]^{cn}_{aol}$ as well).

The await statement in rule ND-AwaitTrue is skipped because there is a value v (which is not η) in the buffer of the stream f at the position determined by the cursor of x. By rule ND-AwaitFalse, the process querying the stream f will be suspended since the cursor of f denotes the empty position in the buffer (denoted by $\perp$). By the semantics, it is not difficult to see that this position will contain either a value v or the termination token η. By the rule ND-AwaitTerminate, the *finished* block s_1 is selected for execution, as the cursor of f points at a position which contains η.

The rule ND-GetTrue assigns to the variable x the value v (which is not η) in the stream buffer from the position determined by the cursor of y, and increments the cursor. By ND-GetTerminate, the **`finished`** block s_1 of the statement is selected for execution followed by s_2, as the cursor of variable y points at the terminating token η in the buffer. Note that the state of x and the cursor are not modified. There is no rule for the **`get-finished`** statement when the cursor denotes the empty position in the buffer (i.e., $\perp$) which implies that the active process (and the object) is *blocked.*

(ND-AsyncCall)
$$\frac{\begin{array}{c} o' = \llbracket e \rrbracket_{a \circ l} \quad \overline{v} = \llbracket \overline{e} \rrbracket_{a \circ l} \\ \mathrm{fresh}(f) \quad \mathrm{streamer}(o'.m(\overline{v})) \end{array}}{\begin{array}{c} ob(o, a, \{l | x = \texttt{\textbf{nd}}\ \ e!m(\overline{e}); s\}, q) \\ \to ob(o, a, \{l | x = (f, 0); s\}, q) \quad invoc(o', f, m, \overline{v}) \quad stream(f, \bot) \end{array}}$$

(ND-AwaitTrue)
$$\frac{(f, n) = \llbracket x \rrbracket^{cn}_{a \circ l} \quad v = \mathrm{elem}(u, n)}{\begin{array}{c} ob(o, a, \{l | \texttt{\textbf{await}}\ \ x?\ \texttt{\textbf{finished}}\ \{s_1\}; s_2\}, q) \quad stream(f, u) \\ \to ob(o, a, \{l | s_2\}, q) \quad stream(f, u) \end{array}}$$

(ND-AwaitFalse)
$$\frac{(f, n) = \llbracket x \rrbracket^{cn}_{a \circ l} \quad \bot = \mathrm{elem}(u, n)}{\begin{array}{c} ob(o, a, \{l | \texttt{\textbf{await}}\ \ x?\ \ \texttt{\textbf{finished}}\ \{s_1\}; s_2\}, q) \quad stream(f, u) \\ \to ob(o, a, \{l | \texttt{\textbf{suspend}}; \texttt{\textbf{await}}\ \ x?\ \texttt{\textbf{finished}}\ \{s_1\}; s_2\}, q) \\ stream(f, u) \end{array}}$$

(ND-AwaitTerminate)
$$\frac{(f, n) = \llbracket x \rrbracket^{cn}_{a \circ l} \quad \eta = \mathrm{elem}(u, n)}{\begin{array}{c} ob(o, a, \{l | \texttt{\textbf{await}}\ \ x?\ \texttt{\textbf{finished}}\ \{s_1\}; s_2\}, q) \quad stream(f, u) \\ \to ob(o, a, \{l | s_1; s_2\}, q) \quad stream(f, u) \end{array}}$$

(ND-GetTrue)
$$\frac{(f, n) = \llbracket y \rrbracket^{cn}_{a \circ l} \quad v = \mathrm{elem}(u, n)}{\begin{array}{c} ob(o, a, \{l | x = y.\texttt{\textbf{get}}\ \texttt{\textbf{finished}}\ \{s_1\}; s_2\}, q) \quad stream(f, u) \\ \to ob(o, a, \{l | x = v; y = (f, n+1); s_2\}, q) \quad stream(f, u) \end{array}}$$

(ND-GetTerminate)
$$\frac{(f, n) = \llbracket y \rrbracket^{cn}_{a \circ l} \quad \eta = \mathrm{elem}(u, n)}{\begin{array}{c} ob(o, a, \{l | x = y.\texttt{\textbf{get}}\ \texttt{\textbf{finished}}\ \{s_1\}; s_2\}, q) \quad stream(f, u) \\ \to ob(o, a, \{l | s_1; s_2\}, q) \quad stream(f, u) \end{array}}$$

Figure 4.9: Operational semantics of non-destructive streams

$$\frac{\overset{(\text{F-AwaitTrue})}{f = [\![e]\!]_{aol}^{cn}}}{\begin{array}{c} ob(o, a, \{l|\textbf{await}\ \ e?\ \textbf{finished}\ \{s_1\}; s_2\}, q) \quad fut(f, v) \\ \rightarrow ob(o, a, \{l|s_2\}, q) \quad fut(f, v)\end{array}}$$

$$\frac{\overset{(\text{F-AwaitFalse})}{f = [\![e]\!]_{aol}^{cn}}}{\begin{array}{c} ob(o, a, \{l|\textbf{await}\ \ e?\ \ \textbf{finished}\ \{s_1\}; s_2\}, q) \quad fut(f, \perp) \\ \rightarrow ob(o, a, \{l|\textbf{suspend}; \textbf{await}\ \ e?\ \textbf{finished}\ \{s_1\}; s_2\}, q) \quad fut(f, \perp)\end{array}}$$

$$\frac{\overset{(\text{F-GetTrue})}{f = [\![e]\!]_{aol}^{cn}}}{\begin{array}{c} ob(o, a, \{l|x = e.\textbf{get}\ \ \textbf{finished}\ \{s_1\}; s_2\}, q) \quad fut(f, v) \\ \rightarrow ob(o, a, \{l|x = v; s_2\}, q) \quad fut(f, v)\end{array}}$$

Figure 4.10: Semantics of futures as streams

Semantics of Futures as Streams

The type of the value of any expression in ABS at runtime is a subtype of the static type of the expression. The **await** and **get** without **finished** clause can only be applied to futures and Boolean guards, and does exclude the streams. This is guaranteed because **Stream**<T> is not a subtype of **Fut**<T> (discussed in section 4.2.4). Recall that an **await** *without* **finished** clause on a stream is only a syntactic sugar for the one *with* the clause where the following block is empty. Different operational semantics of destructiveness and non-destructiveness does not affect the type system.

In order to support the subtyping relation between $\mathsf{stream}\langle T\rangle$ and $\mathsf{fut}\langle T\rangle$ in the operational semantics, as reflected in the type system, we need an extra set of semantic rules, where a future variable appears as the parameter of **await-finished** and **get-finished** statements. This set is presented in Figure 4.10. In these rules, only the cases are specified where the future contains a value v or not (empty stream). A resolved future is treated as an infinite stream of the same value v. Therefore, termination of future is not defined. The rule names are prefixed with `F` to denote that future appears as a stream.

We can prove on the basis of the operational semantics in a standard manner that all program executions are type-safe, and in our case this additionally ensures proper use of the data streams. This additionally amounts to ensuring that the **await-finished** and **get-finished** constructs are applied at runtime only to the data streams (and futures) and that the **yield** operation is only applied to the context of a streaming method.

4.2.6 Discussion on Buffer Size and Garbage Collection

The buffer of streams can grow indefinitely according to the above semantics. For practical purposes, however, we must take into consideration the finite nature of computer memory. This can be addressed by a different definition of the buffer which is bounded to a maximum size m. The semantics of a successful write operation to a bounded buffer requires a new premise where the buffer size is strictly less than m. If the buffer is full, on the other hand, different design decisions can be made. For instance writing to a full buffer can be blocking, i.e., the process is blocked until the buffer size is less than m, or it is non-blocking but signals the process about the failure. A successful destructive read operation, on the other side, decrements the buffer size allowing the buffer to shrink. However this is not the case for non-destructive streams, as the non-destructive read cannot change the buffer size since the data will possibly be read by other cursors. Hence, we need a garbage collection mechanism (GC) for non-destructive streams.

By definition, destructive streams do not cause any garbage. However, we can have a definition of garbage for non-destructive streams. In this section, garbage means a data element in the buffer of a non-destructive stream, which is read by *all* the *existing* cursors. In what follows, we define a GC that is executed periodically. It first obtains all the existing cursors in the system and then collects the garbages accordingly. Some of the cursors can be obtained from the immediate value of a variable, while other cursors can be wrapped with an outer future or stream in a nested way. For instance, if the future variable $x : \mathsf{fut}\langle\mathsf{stream}\langle T\rangle\rangle$ is resolved can possibly contain a cursor. To include these cursors in the GC, first we need some definitions. Let type T denote either a primitive type P (a type that is not a future nor a stream) or a non-primitive type N as follows:

$$\begin{aligned} T &::= \; P \mid N \\ N &::= \; \mathsf{stream}\langle T\rangle \end{aligned}$$

For notational convenience we rewrite the type $\mathsf{fut}\langle T\rangle$ to $\mathsf{stream}\langle T\rangle$. We also rewrite a run-time future object to a destructive stream with at most one element so that it can be typed as a stream. Therefore a future $\mathit{fut}(f, \perp)$ is rewritten to $\mathit{stream}(f, \perp)$ and $\mathit{fut}(f, v)$ to $\mathit{stream}(f, v\eta)$ in cn. The following algorithm obtains the set of all cursors in the system, based on which it marks the garbage:

1. for each object $(o, a, \{l_0|s_0\}, \{\{l_1|s_1\}, .., \{l_k|s_k\}\})$ in the system, the set of cursors that can be obtained in object o: $\mathit{cursor}_o = \{\mathit{cursor}_o([\![x]\!]^{cn}_{aol} : T) \mid x \in a \cup \bigcup_{i=0}^{k} l_i\}$ where $\mathit{cursor}_o(v : T)$ returns all the existing cursors obtained from value v of type T in object o. The set of all the cursors existing in the system: $\mathit{cursors} = \bigcup \mathit{cursor}_o$. Note that for simplicity we assume there is no name conflict of variable names in the mappings.

2. for each stream identity f in the system, $\mathit{min}_f = \mathit{min}(\{n \mid (f,n) \in \mathit{cursors}\})$, where $\mathit{min}(S)$ returns the smallest number in a set S of numbers.

3. for each $\mathit{stream}(f,u)$ in the system, all the data elements in u with the index less than min_f are garbage and must be collected.

For simplicity we use $v : T$ for typing value v instead of using the formal run-time type system. Below we define $\mathit{cursor}^n_o(v : T)$ inductively, where n is the number of times the term stream appears in the type T of the value v, e.g., n is 0, 1, and 2 for the type P, $\mathsf{stream}\langle P\rangle$ and $\mathsf{stream}\langle\mathsf{stream}\langle P\rangle\rangle$, respectively. Note that n is finite, as the type of a variable is a string with a finite length.

Base case: a cursor can neither be obtained from a value with a primitive type (step 0), nor a value that refers to a destructive stream or a future of a primitive type P (step 1). Whereas one cursor can be obtained from a value that refers to a non-destructive stream of a primitive type P (step 1).

$$\begin{aligned}
\mathit{cursor}^0_o(v : P) &= \emptyset \\
\mathit{cursor}^1_o(f : \mathsf{stream}\langle P\rangle) &= \emptyset & \text{where } \mathit{stream}(f,u) \in \mathit{cn} \\
\mathit{cursor}^1_o((f,n) : \mathsf{stream}\langle P\rangle) &= \{(f,n)\} & \text{where } \mathit{stream}(f,u) \in \mathit{cn}
\end{aligned}$$

Inductive step: the induction hypothesis is that $\mathit{cursor}^n_o(v : N)$ returns all the cursors obtained from the value v of type N where $n = k$. Below we show how we obtain the cursors obtained from a value for $n = k+1$ using the hypothesis:

$$\begin{aligned}
\mathit{cursor}^{k+1}_o(f : \mathsf{stream}\langle N\rangle) &= \bigcup_{v_i \in s_n(u)} \mathit{cursor}^k_o(v_i : N) \\
&\quad \text{where } \mathit{stream}(f,u) \in \mathit{cn} \\
\mathit{cursor}^{k+1}_o((f,n) : \mathsf{stream}\langle N\rangle) &= \bigcup_{v_i \in s_n(u)} \mathit{cursor}^k_o(v_i : N) \cup \{(f,n)\} \\
&\quad \text{where } \mathit{stream}(f,u) \in \mathit{cn}
\end{aligned}$$

where $s_n(u)$ denotes a set of elements in the buffer u of $\mathit{stream}(f,u)$ with index greater or equal to n, except special elements $\perp$ and η.

In order for the above algorithm to work, every data element in the buffer u must have an absolute index starting from zero for the first element added to the buffer. Recall that, by the semantics of ABS, every synchronous and asynchronous method call forms a (suspended or active) process in the called object which is denoted by $\{l_i|s_i\}$. Hence, the step 1 covers all the variable assignments in an object (cursor_o) and subsequently in the whole system ($\mathit{cursors}$). Also note that there is no need to distinguish the cursors by their variable names or their processes or objects. The

$$\text{(T-S{\scriptsize TREAM})}\quad \frac{\begin{array}{c}\Delta(f) = \mathsf{stream}\langle T\rangle \\ \forall i \in [1..n].(val_i \not\in \{\bot, \eta\} \Rightarrow \Delta(val_i) = T)\end{array}}{\Delta \vdash_R \mathit{stream}(f, (val_1, ..., val_n))\ \ \mathsf{ok}} \qquad \text{(T-S{\scriptsize TATE}S{\scriptsize TREAM})}\quad \frac{\begin{array}{c}\Delta(val) = (\mathsf{stream}\langle T\rangle, \mathsf{Nat}) \\ \Delta \vdash_R v : \mathsf{stream}\langle T\rangle\end{array}}{\Delta \vdash_R \mathsf{stream}\langle T\rangle\ \ v\ \ val\ \mathsf{ok}}$$

$$\text{(T-I{\scriptsize NVOC}S{\scriptsize TREAM})}\quad \frac{\Delta(f) = \mathsf{stream}\langle T\rangle \quad \Delta(\overline{v}) = \overline{T} \quad \mathrm{match}(m, \overline{T} \to \mathsf{stream}\langle T\rangle, \Delta(o))}{\Delta \vdash_R \mathit{invoc}(o, f, m, \overline{v})}$$

Figure 4.11: The typing rules of streams for run-time configurations

only relevant aspect of the cursor for GC is that there exists at least one cursor that points at a specific index of the buffer.

4.3 Subject Reduction for the Extended ABS

A *run* is a sequence of transitions from an *initial state* based on the rules of the operational semantics, where *initial state* consists of $ob(\text{start}, \epsilon, p, \emptyset)$, an initial object, start, with only one process p that corresponds to the main block of the program. The subject reduction for ABS is already proven in [48], namely, it is shown that a run from a well-typed initial configuration will maintain well-typed configurations, particularly, the assignments preserve well-typedness and method bindings do not give rise to the **`error`** process. In this section, we aim to extend the proof for the ABS subject reduction theorem to also include the notion of stream as specified in this chapter.

The typing context for the run-time configurations Δ extends the static typing context Γ with typing dynamically created values (entities created at run-time), namely, object and future identifiers. Let $\Delta \vdash_R cn$ **ok** express that the configuration cn is well-typed in the typing context Δ. The typing rules for run-time configurations are defined for ABS and extensively discussed in [48]. The newly added rules for typing streams are shown in Figure 4.11. By T-Stream, the stream f is of type $\mathsf{stream}\langle T\rangle$ if the buffer only contains values of type T or the special tokens η and $\bot$. By T-StateStream, a variable v that refers to a stream val and provide non-destructive access to it is well-typed. Nat denotes the type of natural numbers. The type of val is a pair of the stream type and a Nat that holds the cursor to the stream. The rule T-InvocStream allows the return type of an asynchronous method invocation to be a stream as well.

In [48] (1) it is proven that the initial object corresponding to the main block of a well-typed program is well-typed and also (2) it is shown that the well-typedness of runtime configuration is preserved by reductions (Theorem 1). The proof for (1) also applies here. We only need to extend the proof for (2) with respect to the new transition rules introduced in section 4.2 as follows.

Theorem 1 (Subject Reduction)**.** *If $\Delta \vdash_R cn$ **ok** and $cn \rightarrow cn'$, then there is a Δ' such that $\Delta \subseteq \Delta'$ and $\Delta' \vdash_R cn'$ **ok**.*

Proof. The proof is by induction over the defined transition rules in the operational semantics. We assume objects, futures, streams and messages not affected by a transition remain well-typed, and are ignored below. The auxiliary function $match(m, \overline{T} \rightarrow \mathsf{stream}\langle T\rangle, T')$ checks if a method m with $\overline{T} \rightarrow \mathsf{stream}\langle T\rangle$ is provided by the interface T'.

- *Process Suspension.* It is immediate that the rules D-AwaitTrue, ND-AwaitTrue,F-AwaitTrue, D-AwaitFalse, ND-AwaitFalse, F-AwaitFalse D-AwaitTerminate, ND-AwaitTerminate, D-GetTerminate and ND-GetTerminate preserve the well-typedness.

- Yield. By assumption, we have $\Delta \vdash_R ob(o, a, \{l|\mathbf{yield}\ e; s\}, q)$ **ok**, $[\![e]\!]_{aol} = v$ and $\Delta \vdash_R stream(f, u.\bot)$ **ok**. Obviously, $\Delta \vdash_R ob(o, a, \{l|s\}, q)$ **ok**. Since $l(\text{destiny}) = f$ and l is well-typed, we know that $\Delta(\text{destiny}) = \Delta(f)$. Let $\Delta(f) = \mathsf{stream}\langle T\rangle$. By T-Yield, $\Delta \vdash_R e : T$ and subsequently $\Delta(v) = T$, so $\Delta \vdash_R stream(f, u.v.\bot)$ **ok**.

- ReturnStream. By assumption, we have $\Delta \vdash_R ob(o, a, \{l|\mathbf{return}; s\}, q)$ **ok**, and $\Delta \vdash_R stream(f, u.\bot)$ **ok**. Obviously, $\Delta \vdash_R ob(o, a, \{l|s\}, q)$ **ok** and $\Delta \vdash_R stream(f, u.\eta)$ **ok**.

- D-AsyncCall. Let $\Delta \vdash_R ob(o, a, \{l|x = e!m(\overline{e}); s\}, q)$ **ok**. We first consider the case $e \neq \mathbf{this}$. By T-AsyncStream, we may assume that $\Delta \vdash e!m(\overline{e}) : \mathsf{stream}\langle T\rangle$ and by T-Assign that $\Delta(x) = \mathsf{stream}\langle T\rangle$. Therefore, $\Delta \vdash e : T'$ and $\Delta \vdash \overline{e} : \overline{T}$ such that $match(m, \overline{T} \rightarrow T\ \mathsf{stream}, T')$. Assume that $[\![e]\!]_{aol} = o'$ and let $\Delta(o') = C$ for some class C. Based on [48], there is a Δ' such that $\Delta' \vdash_R [\![e]\!]_{aol} : T'$ and $\Delta'(o') = C$, so $C \preceq T'$. By assumption class definitions are well-typed, so for any class C that implements interface T' we have $match(m, \overline{T} \rightarrow T\ \mathsf{stream}, C)$. Also $[\![\overline{e}]\!]_{aol}$ similarly preserves the type of $\overline{e}$. Let $\Delta'' = \Delta'[f \mapsto \mathsf{stream}\langle T\rangle]$. Since $fresh(f)$ we know that $f \notin dom(\Delta')$, so if $\Delta' \vdash_R cn$ **ok**, then $\Delta'' \vdash_R cn$ **ok**. Since $\Delta' \vdash e!m(\overline{e}) = \Delta''(f)$, we get $\Delta'' \vdash_R ob(o, a, \{l|x = f; s\}, q)$ **ok**. Furthermore, $\Delta'' \vdash invoc(o', f, m, \overline{v})$ **ok** and $\Delta'' \vdash_R stream(f, \bot)$ **ok**. The case $e = \mathbf{this}$ is similar, but uses the class of **this** directly for the match (so internal methods are also visible).

- ND-AsyncCall. Let $\Delta \vdash_R ob(o, a, \{l \mid \mathbf{nd}\ x = e!m(\overline{e}); s\}, q)$ **ok**. The argument is similar to the above case, but we get $\Delta'' \vdash_R ob(o, a, \{l|x = (f, 0); s\}, q)$ **ok** as the consequence, in addition to $\Delta'' \vdash invoc(o', f, m, \overline{v})$ **ok** and $\Delta'' \vdash_R stream(f, \bot)$ **ok**.

- D-GetTrue. By assumption, $\Delta \vdash_R ob(o, a, \{l|x = e.\mathbf{get}\ \mathbf{finished}\{s_1\};$ $s_2\}, q)$ **ok**, $\Delta \vdash_R stream(f, v.u)$ **ok**, and $[\![e]\!]_{aol} = f$. Let $\Delta(f) = \mathsf{stream}\langle T\rangle$.

Consequently, $\Delta \vdash_R e.$**get finished**$\{s_1\} : T$ and $\Delta(v) = T$, so $\Delta \vdash x = v$, $\Delta \vdash_R ob(o, a, \{l|x = v; s\}, q)$ **ok** and $\Delta \vdash_R stream(f, u)$ **ok**. A similar argument applies for F-GetTrue where f is the identity of a future object $fut(f, v)$.

- ND-GetTrue. By assumption, $\Delta \vdash_R ob(o, a, \{l|x = y.$**get finished**$\{s_1\}; s_2\}, q)$ **ok**, $\Delta \vdash_R stream(f, u)$ **ok**, $[\![y]\!]_{a \circ l} = (f, n)$ and $\mathrm{elem}(u, n) = v$. Let $\Delta(f) = \mathsf{stream}\langle T\rangle$. Consequently, $\Delta \vdash_R y.$**get finished**$\{s_1\} : T$ and $\Delta(v) = T$, so $\Delta \vdash x = v$, $\Delta \vdash_R ob(o, a, \{l|x = v; s\}, q)$ **ok**, and $\Delta \vdash y = (f, n + 1)$ **ok**.

□

4.4 Data Streams in Distributed Systems

In [10] a scalable distributed implementation of the ABS language is described. In this section we adapt our proposed notion of data streams in ABS to reduce the possible overhead of data steaming in a distributed setting.

To this aim, each streaming method is enabled to package the return values, that is, the method populates its return stream buffer possibly not once per value, but once per sequence of values. The package size can be specified explicitly as a parameter or can be selected based on the underlying deployment, e.g., it can be equal to the packet size of the TCP/IP technology involved. As such the number of packets to be transferred through the network is minimized.

There are two conditions when the package is streamed *before* its size is equal to the pre-specified package size: 1) when the streaming method terminates; 2) when the streaming method cooperatively releases control. The first condition is obvious, while the second prevents a specific kind of deadlock configuration. In general, ABS programs may give rise to deadlocks (see [39] for a discussion of deadlock analysis of ABS programs). However the notion packaging data streams should not give rise to additional deadlock possibilities.

The above second condition prevents the following kind of deadlock situation. Note that *package size* $= n$ means that the number of yielded values needs to be equal to n, so that they are streamed as a package, except for the last package where the size may be less than n. Suppose there are two objects o_1 and o_2 in the run-time configuration where o_1 executes an active process which corresponds to method m_1 given by

$m1()\{$`r=o2!m2();` **await** `r?; o2!satisfier();`$\}$

and the specification of the streaming method m_2 is an active process p in object o_2 given by

$m2()\{$**yield** `x;` **await** `e;` **yield** `y;`$\}$

Furthermore, suppose the method *satisfier* in o_2 changes the object state so that the expression e (which is `False` initially) evaluates to `True`. It is not difficult to see that for all $n \geqslant 2$, where n is the package size of the stream, the run-time configuration is deadlocked. The reason is that the first yielded value is not streamed before p releases control, as the package size is smaller than n. The deadlock possibility can be generalized to a category of programs where a streaming method releases control before it communicates the values which are yielded. The solution is that the package with the size smaller than n is streamed, before the process cooperatively releases control or blocks.

4.5 Implementation

In this section, we present a prototype implementation of future-based data streams as an API written in ABS. This API (see Figure 4.12) can be used to simulate the semantics of data streaming in ABS itself. The implementation details of the API can be found online[1].

As discussed in section 4.2, the `Stream<T>` datatype is parametrically polymorphic in its contained values of type T. The original ABS specification, however, offers besides parametric polymorphism also *subtype* polymorphism, through its interface types. In general, when defining and implementing languages with support for subtype-polymorphism, often the issue of *variance* arises: where in the code it is allowed (i.e. type-safe) to upcast to a supertype or downcast to a subtype. For example, given a subtype relation (`T` is subtype of `U`), a structure `S` is called covariant if `S<T>` is safe to "upcast" to `S<U>`; contraviarant if safe to "downcast" `S<U>` to `S<T>`; invariant if none of the above two hold, i.e. subtype polymorphism cannot be used for this structure, but other methods of polymorphism (e.g. parametric) perhaps can. In practice, the "rule of thumb" suggests that structures which are exclusively read-only (i.e. immutable) are allowed to be covariant, structures that are written-only (e.g. log files) contravariant, and structures that are read-write must be invariant.

The extension of ABS with stream that we describe in this chapter, strictly separates at the syntax level the role of the producer of values (write to the stream structure) with the role of the consumer (read from the stream). Since the producer can only append (produce) new values to the stream and not alter (mutate) past values, from the sole point of the consumer the stream structure seems as "immutable" (covariant). In this sense, a consumer holding a variable of type `Stream<T>` should be allowed to upcast it to type `Stream<U>`. Conversely, the producer is allowed to `yield` values of subtype `T`, if the method call's return type is typed as `Stream<U>`. As such, at the surface level (syntax and type system) it is acceptable for the `Stream` structure to be treated as covariant; however, at the implementation level it still re-

[1] `https://github.com/kazadbakht/ABS-Stream/blob/master/lib/Streams.abs`

mains a challenge on how to guarantee type safety at the host language (in our case, Haskell).

The Haskell language has parametric polymorphism but lacks built-in support for subtype polymorphism; for this reason, the ABS-Haskell backend compiler generates dynamic "upcasting" function calls where needed. However, this technique cannot be applied as well with Haskell's builtin *vector* datatype, which is a low-level built-in structure that cannot be made covariant or contravariant since it has been fixed-byte allocated in memory heap upon creation. For this reason, and also the fact that arrays are in general a mutable (read-write) data structure, the vectors in ABS (borrowed from Haskell) are treated as invariant. Since the implementation of streams in ABS relies currently on vectors, there is the practical limitation of having the `Buffer` type to be invariant. Similarly the `Stream` and `Fut` datatypes are treated as invariant, because the ABS-Haskell backend treats each future `Fut<T>` as a pointer to a vector size-1 stored in the heap that holds the value of `T`. Based on this practical limitation, the `Stream<T>` datatype introduced in this extension to ABS is treated as subtype-invariant, with support for parametric polymorphism.

The API is semantically compliant with the semantics of data streams defined in this chapter: The method that yields to a stream is separated from the access mode of readers to the stream (i.e., either destructive or non-destructive). Every reader has access to a stream via an instance of either `Dref` or `NDref` for destructive or non-destructive access mode, respectively. Furthermore a stream variable (that refers to an instance of `Dref` or `NDref`) is only typed by the `Stream` interface, abstracting from the underlying access mode.

The interface `Buffer<T>` is implemented by the class `CBuffer`. The FIFO buffer (an instance of `CBuffer`) is implemented by a vector whose elements are of type `Maybe<T>`, namely, each of which contains either a value (`Just(v)` where v is of type `T`) or `Nothing`. A position in the vector can have three different states: It contains `Just(v)` (a value `v` that can be read), `Nil` (the position is empty and will be filled), and `Nothing` (a token of type `Maybe<T>`) that denotes termination of the stream. The interface `Buffer<T>` provides the methods `yield()` and `terminate()` to the streaming method in order to write to the buffer and to explicitly terminate the stream of data values, respectively. The termination enqueues `Nothing` to the buffer and is meant to be the last statement in the definition of the streaming method (to simulate the terminating **`return`**). A stream maintains a global index `wrt` to the buffer which denotes the position where the next yielded value is written. It is incremented by every time calling `yield`. In destructive read, the `CBuffer` maintains a global index (i.e., `rd`) to the buffer for all the readers of the stream, whereas in non-destructive read, every reader (i.e., `NDref` instance) maintains a local index (i.e., `cursor`) to the buffer.

The reader can read from a stream by asynchronously calling `pull()` on the `Stream` object that returns a *future* representing the next data value, whether resolved or not. The operation `pull` is overridden in `Dref` and `NDref` for de-

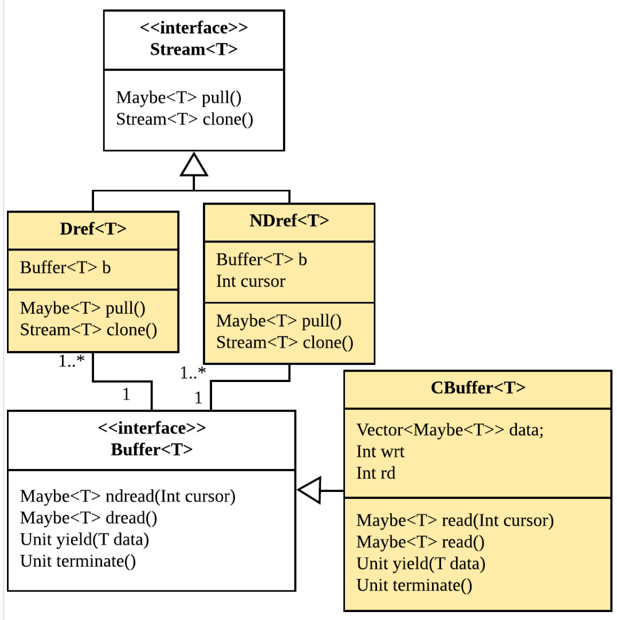

Figure 4.12: Class diagram of ABS Stream library

```
(1) Maybe<T> dread() {
(2) Int temp = rd;
(3) rd = rd + 1;
(4) await (buffer[temp] != Nil);
(5) return buffer[temp]; // is not null
(6) }

(7) Maybe<T> ndread(cursor) {
(8) await (buffer[cursor] != Nil);
(9) return buffer[cursor]; // is not null
(10)}
```

Figure 4.13: Destructive and non-destructive read in `CBuffer`

structive and non-destructive read from the buffer, respectively. The former calls `dread()` method of the `Buffer` which returns the first valid element in the vector, indicated by the index `rd` in buffer, and increments `rd`. Whereas the latter calls `ndread(cursor)` of the `Buffer` where the cursor is a field of the `NDref`, which returns the element indicated by the index `cursor` in the vector. The implementation of `dread()` and `ndread(cursor)` is given in Figure 4.13 where **`await`** at lines (4) and (8) cooperatively release control until the condition (indicating whether the buffer element has been produced) holds.

Also the method `clone` is used to copy a non-destructive stream object, a new instance of `NDref` which has a reference to the same stream but a new cursor which is initialized with the value of the cursor of the original object. For destructive streams, the method only returns the reference to **`this`** which is of type `Dref`.

Awaiting the future resulting from calling `pull()` queries the availability of next data value. Therefore, statement **`await`** `r?` **`finished`** {`S`} is expressed in the library as follows:

```
(1) f = r!pull(); // a future f to the target data value
(2) await f?; // awaits if the future f is not resolved yet
(3) m = f.get; // gets the resolved data value
(4) if (m == Nothing) // Nothing is the special token denoting the termination
(5) {S}
```

where `r` is a reference to a stream object and `S` is a block of statements. This can either give rise to the release of control in case the data is not available (line 2) or to skip otherwise. The variable `m` is of type `Maybe<T>` which contains either the value `v`, denoted by `Just(v)`, where `v` is of type `T`, or `Nothing`.

Similarly, statement `x = r`**`.get finished`** {`S`} can be expressed using the library as follows:

```
(1) f = r!pull(); // a future to the target data value
(2) m = f.get; // gets the resolved data value
(3) case (m) {
(4) Nothing => {S} // "Nothing" is the special token denoting the termination
(5) Just(v) => {x = v} // the value v is assigned to x
(6) }
```

In line 2, the object running this process blocks until the the data value is written to the future `f`.

The keyword **`nd`** is implemented in the API by a Boolean argument passed to the called streaming method. The argument specifies whether the return object of the streaming method to be an object of class `Dref` or class `NDref`.

The following snippet shows how the library is used to stream *integer* data values. The streaming method `m` instantiates a stream, delegates yielding values to the stream asynchronously to an auxiliary method `m2`, and returns the stream to the caller. Note that `m` sends the same list of parameters it receives to `m2`.

```
// caller :
// False means the return stream is non-destructive
Fut<Stream<Int>> f = o!m(False, ...);
Stream<Int> r = f.get; // r is a reference to the stream
// reading from the non-destructive stream r
...

// callee
Stream<Int> m(Bool isDestr, ...){
      Buffer<Int> b = new Cbuffer();
      // b is the return stream which is filled by m2
      this!m2(b, ...);
      // isDestr determines the access mode to the stream
      // i.e., destructively or non-destructively
      if (isDestr)
            return new Dref<Int>(b);
      else
            return new NDref<Int>(b, 0);
}

// the implementation of the callee
Unit m2(Buffer<Int> b, ...) {
      Int x ;
      ...
      b!yield(x); // yield a value
      ...
      b!terminate(); // termination token
}
```

Remark. In the above-mentioned API, having multiple readers for one stream may result in a performance bottleneck, as the buffer object itself queries the availability of data item to be returned for every `pull` request (via `dread` or `ndread`). Alternatively, such availability check can be delegated to the reader's `pull` method itself. However in the current design, doing the checks of line (4) and (8) in Figure 4.13 in their corresponding `pull` definitions give rise to busy-wait polling. The key feature that enables delegating the check without busy-wait polling is the data type `Promise<T>`. A promise is of this type is either contains a data value of type `T` (resolved) or not. An unresolved promise p can be resolved by *p.give(v)* with some data value v. Similar to futures, *get* and *await* operations can also be applied to promises.

By converting the type of the buffer vector from `Maybe<T>` to `Promise<Maybe<T>>`, the methods `dread` and `ndread` can immediately return to the reader's `pull` request the promise object in the vector that holds the expected value, which is either already resolved and can be retrieved or will be resolved in the future. In this new design, the `ndread` in `CBuffer` only returns the promise without availability check as follows:

```
Promise<Maybe<T>> ndread(cursor) {
return buffer[cursor];
}
```

And `pull` method in `NDref` which checks the availability is given as follows:

```
(1) Maybe<T> pull() {
(2) Fut<Promise<Maybe<T>>> f = b!ndread(cursor); // cursor is a field in NDref
(3) Promise<Maybe<T>> p = f.get;
(4) await p?; // availability check is moved to pull
(5) Maybe<T> result = p.get;
(6) case (result) {Just(v) => cursor = cursor + 1;}
(7) return result;
(8) }
```

In line (6), if `result` is equal to `Nothing` then `cursor` is not incremented, such that the next `pull` requests to this object result in `Nothing`. Similar changes apply to `pull` in `Dref` and `dread()` in `CBuffer` for destructive read, except the index `rd` which is updated in `dread`.

An implementation similar to the example in Figure 4.2 using the above API is provided online[2]. We also ran the implementation on a PC which was an Intel Core i7-5600U 2.60GHz × 4 with 12GB RAM, and 64-bit Ubuntu 16.04 LTS as the

[2]`https://github.com/kazadbakht/ABS-Stream/blob/master/Examples/map\_reduce.abs`

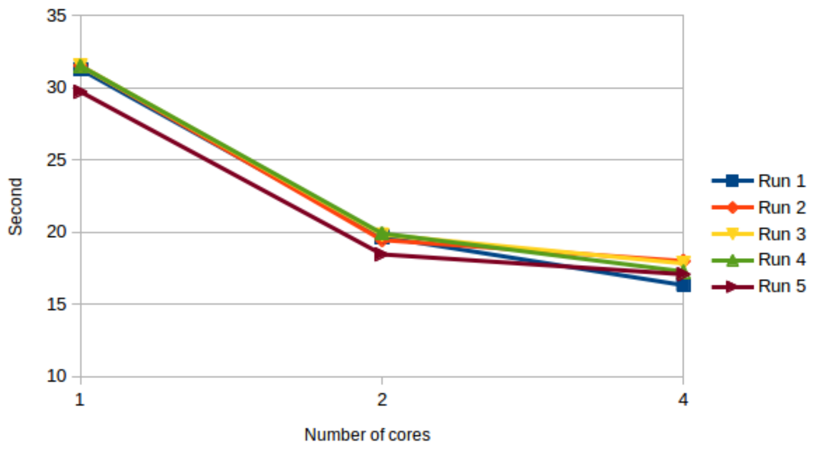

Figure 4.14: Execution time of parallel map-reduce program for 10^6 data values

operating system. In Figure 4.14, we represent the time measured for execution of the program for different number of parallel processors in 5 runs. On average for this specific implementation, we observed that for two cores we achieve 1.59× speedup compared to one core, and for four cores 1.13× speedup compared to two cores.

Lower speedup achieved for higher number of processors, among other reasons, stems from the fact that a stream is a shared resource where parallel access and yielding values to it in a safe manner is limited. This is confirmed experimentally, by adding more workload on the reader per data value that it reads from the stream. As such, the access rate to the stream becomes low enough such that the stream is not a performance bottleneck. With this modification, we could achieve up to 1.41 speedup for four cores compared to two cores.

The current implementation does not feature garbage collection of streams: produced data values are stored in a vector which dynamically (at-runtime) grows indefinitely (until memory exhaustion). This choice was made for a separation of concerns. This orthogonal issue of garbage collection can be trivially solved in the case of destructive streams: all produced values before the global index can be considered as garbage. A future implementation should automatically reclaim the space for such values and appropriately resize (shrink) the vector. In the case, however, of non-destructive streams, some extra bookkeeping and communication is involved to have safe, distributed garbage collection of streams. One possible implementation would require storing at the producer's side a global (system-wide) minimum of all the readers' local cursors. Besides this bookkeeping of the producer, once a reader forwards a `Stream<T>` to another ABS process (local or remote), it involves notifying the producer about the local minimum of the new reader process. Furthermore, in case of a real distributed system, the producer should monitor the quality of the network connection to every reader, otherwise it runs the risk of memory leaking from a dropped connection to a reader.

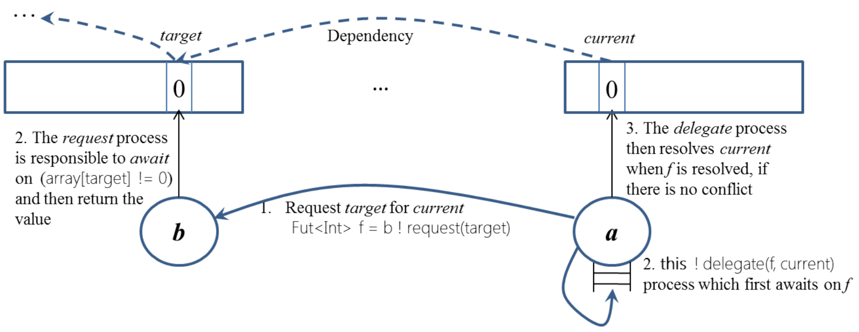
...
target
Dependency
current
0
...
0
2. The *request* process is responsible to *await* on (array[target] != 0) and then return the value
3. The *delegate* process then resolves *current* when *f* is resolved, if there is no conflict
1. Request *target* for *current*
Fut<Int> f = b ! request(target)
b
a
2. this ! delegate(f, current) process which first awaits on *f*

Figure 4.15: Resolving dependencies in Distributed PA using cooperative scheduling

4.6 Case Study

In this section, we use data streams in the ABS model of the distributed PA proposed in chapter 3. Figure 4.15 illustrates the high-level scheme for the unresolved dependencies proposed in chapter 3, using the notion of an active object, future, and cooperative scheduling in ABS.

In above scheme, the current, unresolved slot which belongs to the active object a requires the value of the unresolved target slot which belongs to the the active object b. To this aim, the object a asynchronously calls the *request* method of the object b, and delegates the resulting future as a suspended process in its queue, so that the active process continues with the rest of its partition. On the other side, the request awaits on a Boolean condition which checks if the target is resolved and returns the value. Finally, the *delegate* method which awaited on the future, gets the future value and processes it.

4.6.1 Incorporating Data Streams

The generation of distributed PA-based graphs as described above is fairly high-level and intuitive at the modeling level. However, the number of messages and return values communicated among the active objects poses a considerable overhead. Packaging the requests and the corresponding return values can considerably improve the performance of the run-time system.

In the distributed scheme in Figure 4.15, the request is sent per each required target slot, which is too fine-grained. Instead, we propose a modification of the algorithm so that the requests for the target values located on the same active object are sent together as a package of requests via one message, and the returning values are received via a stream with packaging capability. An experimental validation of a scalable distributed implementation of our model that utilizes streams is presented in [11] which is based on Haskell. It shows a significant performance improvement for the model compared the one presented in [10] (chapter 3).
Resolving the dependencies in the modified approach is shown in Figure 4.16. For

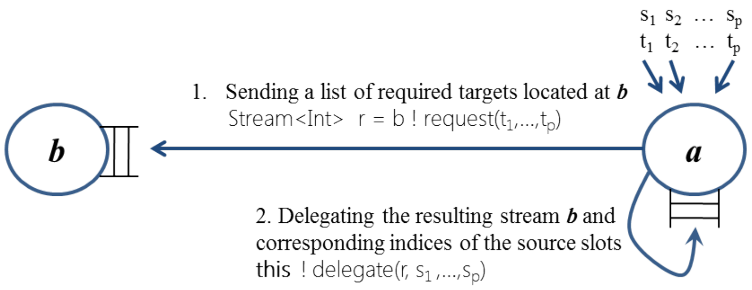
s_1 s_2 … s_p
t_1 t_2 … t_p
1. Sending a list of required targets located at ***b***
Stream<Int> r = b ! request(t_1,…,t_p)
b
a
2. Delegating the resulting stream ***b*** and corresponding indices of the source slots
this ! delegate(r, s_1,…,s_p)

Figure 4.16: The modified approach using destructive streams

all $i \in [1, p]$, each pair (s_i, t_i) represents a request from object a to object b, where s_i represents the index of an unresolved slot belonging to the partition hosted by a, and t_i represents its corresponding slot belonging to the partition hosted by b. The value obtained from each t_i is used to resolve the unresolved slot s_i. Assuming p is the package size, the list $[t_1, .., t_p]$ is sent to b as a package of requests, and the requests process returns corresponding values per each t_i via a stream r (e.g., ***yield*** *array*$[t_i]$). Figure 4.17 illustrates abstract ABS code for *requests* which streams the values, and for *delegates* which receives them.

4.7 Related Work

There already exists a variety of different programming constructs for streaming data in different programming languages like Java, and software frameworks for processing big data like Apache Hadoop and Spark.

Asynchronous generators specified in [66] 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 multi-threaded model of concurrency, where asynchrony is provided by spawning a thread for a method call.

Akka Streams [70] provides 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 *channel* as a means of communication, inspired by the model of *Communicating Sequential Processes* (CSP). For instance, Go language and *JCSP* [76], which is a library in Java, provide CSP-like elements, e.g., processes (referred to as Goroutines in Go) that communicate via channels by means of read and write primitives. Buffered channels in Go provide asynchronous read (cf. write) when the buffer is not empty (cf. not full). Otherwise the primitives are blocking.

Similarly to asynchronous generators, streaming data as proposed in this chapter

```
Each actor o executes the following in parallel
Unit run(...)
while the whole partition is not yet processed do
    /*
      Resolve the slots. Next pack of unresolved sources = $[s_1, .., s_p]$ from
      the partition belonging to this object, and its corresponding targets =
      $[t_1, .., t_p]$ whose owning partition hosted by some object w are calculated
    */
    Stream<Int> $f = w$ ! requests(targets);
                                        ▷ The stream $f$ is destructive by default
    this ! delegates($f$, sources);

Int stream requests(List<Int> targets)
while targets is not nil do
    Int tar = head(targets);
    targets = tail(targets);
    await ($arr[tar] \neq 0$);
                                        ▷ At this point the target is resolved
    yield arr[tar];

Unit delegate(Stream<Int> r, Int sources)
while True do
    Int val;
    await r? finished {break;}
                                        ▷ Quit the while if $r$ is terminated
    val = r.get;
    Int src = head(sources);
    sources = tail(sources);
    // Use val to resolve arr[src]
```

Figure 4.17: The sketch of the data streaming in the modified approach

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.

The distributed shared memory (DSM) paradigm [34, 57, 72] enables access to a common shared space across disjoint address spaces, where communication and synchronization between processes are enforced through operations on shared data. The notion of *tuple space* was originally integrated at the language level in Linda [37]. The processes communicate via insertion/read/removal of tuples into/from the tuple: `out()` to write a tuple into a tuple space, `in()` to retrieve (and remove), and `read()` to read (without removing) a tuple from it.

Similarly to tuple spaces, the interaction model of streams proposed in this chapter provides time and space decoupling, namely, data producers and consumers can remain anonymous with respect to each other, and the sender of a data needs no knowledge about the future use of that data or its destination (the reference to a stream can be passed around). Also producer-side synchronization decoupling is guaranteed, whereas, the consumer-side decoupling is not provided in tuple-spaces, as the consumers synchronously pull the data. In ABS streams, however, the decoupling is provided at the consumer object level, thanks to the notion of cooperative scheduling. Similarly to tuple space *in*-based and *read*-based communication, the destructive and non-destructive data streams, respectively, can be naturally used to implement *one-of-n* semantics (only one consumer reads a given data), and *one-to-n* message delivery (a given data can be read by all such consumers).

4.8 Future work

We focused on extending the main asynchronous core of ABS with data streams. Other main features of the ABS like *concurrent object groups* (cogs) and deployment components are orthogonal and compatible with this extension. As an example, ABS features *cog* that, in principle, shares a thread of control among its constituent objects, which enables internal synchronous calls. By the nature of data streaming, it is natural to restrict the streaming to the asynchronous method calls.

Another interesting line of work consists of investigating a more efficient GC for non-destructive streams. The approach proposed in this chapter involves a periodic execution of GC that requires gathering information from all actors in the system synchronously which can in practice give rise to a bottleneck. Alternatively, a more efficient GC can be investigated where each stream maintains a table counting the number of cursors to each data element in the buffer.

The ABS with Haskell backend supports real-time programming techniques which allows for specifying deadlines with method invocations. This provides an interesting basis to extend ABS with real-time data streaming which may, as an example, involve timeout on read operations. We also need to extend the various formal analysis techniques (e.g., deadlock detection, general functional analysis based on method contracts) currently supported by the ABS to the ABS model of streaming data discussed in this chapter.