Chapter 7

ABS to Scala Compiler

Together with the runtime presented in Chapter 4, a source-to-source translation from ABS to Scala is provided to support both the object-oriented and functional programming paradigms of ABS. The most important features of the compiler are the correct translation of the core ABS semantics together with the coroutines feature of ABS, the compilation of the Timed-ABS model extension and the translation of the ABS extension that models deployment components and consumption of resources.

7.1 Translating Core ABS and its Extensions to the Proposed Runtime

To provide a compiler for ABS for all the core features discussed in Chapter 3 a lot of the work that already existed for state-of-the-art backends for Maude and Erlang in [absc] was reused. For parsing and type-checking the Core ABS language we used the grammar and type system already in place to obtain the Abstract Syntax Tree(AST) of the program. The syntactical and semantic analysis steps are the same for every backend, the proposed compiler reuses these steps. The code generation step is backend specific and generates Scala code that uses the proposed Java Runtime. This involves using a visitor pattern to traverse the AST and generate code for each tree node. Most of the nodes do not present a difficult task, as interfaces, classes, method signatures or bodies, control statements and expressions all have one-to-one correspondents in Java or Scala. The exception to all these are the algebraic data types and the control statement *case* (discussed in Section 3.2) which provide the functional part of ABS. For this paradigm of ABS the translation must be done to the Scala language discussed in depth section 7.2. However, the rest of the code-generation for the core ABS is one of the main challenges as it has to take into account all of the particularities of ABS discussed in Chapter 3, which are:

- Instantiating objects as actors: all of the objects created using the `new` command behave like actors and need to be separated in their own COGs unless indicated by the `local` option which specifies sharing a COG with the parent object.
- Asynchronous communication: method invocations created using the "!" operator must be created as lambda expressions and passed to the called object/actor.
- Continuations: when release points are met, actors need to know from which point and in what state to resume execution of either the actor itself or a suspended message.
- Cooperative Scheduling: execution flow must be controlled to follow the semantics in Section 3.3 when encountering a `get` or `await` construct.

7.1.1 Objects as Actors in COGs

ABS objects are not just regular JVM objects, and require some changes when they are instantiated. First, all interface declarations need to extend the **Actor** interface and all class declarations need to extend the **Local Actor** default implementation such that newly instantiated objects can behave like actors. Extending this base interface and class allows for the calling of the default constructor that initializes each object with a message queue and a `MainTask` to iterate through it. It also makes available for each object the methods `send`, `spawn` and `getSpawn` to be used for asynchronous communication and cooperative scheduling). A simple layout of an interface and class declaration in ABS is given in Listing 7.1. According to ABS semantics, all declarations need to be comparable, whether they are defined as ADTs or objects, mainly due to the definitions and assumptions made in the ABS Standard Library. With respect to objects, there is no particular rule to comparing objects, so this is left to the default ordering that Scala makes for objects. For this to happen, all parent interfaces need to extend the Scala trait `Ordered` as shown in Listing 7.2.

Listing 7.1: An Interface and Class in ABS

```
1 interface Ping {
2   ...
3 }
4
5 class PingImpl(...) implements Ping {
6   ...
7 }
```

Second, we need to differentiate between the `new` and `new local` constructs. The presence of the `local` option puts the newly created ABS object into the same COG as the object that created it, while the absence of this option will place it a new COG. What this means for the generated code is that the actors have to create new or share the `messageQueue` or `mainTaskIsRunning` variables discussed in Chapter 4 such they can emulate placement on a new COG or the same one. To add this support, the generated code for each class adds a parameter `destCOG` as shown in Listing 7.2. This reference may point to the actor that called `new local` such that the the `messageQueue` or `mainTaskIsRunning` will be shared, otherwise if the reference is null, the variables will be created by the parent constructor. Depending on the location of the new object, its constructor will call moveToCOG(callerofNewLocal) or moveToCOG(null) (line 10) . The call to this method is very important as it needs to be done immediately after calling the super( ) constructor of the LocalActor class. The generation of a statement like **new** local Ping(...); will look as **new** Pingthis,...);)

Listing 7.2: COG and DC Assignment

```
1  trait Ping extends Actor with Ordered[Actor] {
2    ...
3  }
4
5  class PingImpl(var destCOG : LocalActor,var destDC : Actor,...)
6      extends LocalActor with Ping {
7
8   //constructor
9   {
10    moveToCOG(destCOG);
11    setDc(destDC);
12  }
13 }
```

Deployment Components Very closely related to COG assignment is the option to generate code with resource model support. This generation is optional and can be activated as an argument in the command line. To achieve this, the compiler automatically generates a second parameter for every ABS class definition (`destDC`). Again this parameter determines on which resource the newly created object will be placed through calling setDC(targetDC) (line 11). An ABS `new` construct with resource model support can optionally have an annotation such as in Listing 3.12. This construct looks like [DC: dc] Worker w = **new** CWorker( ) In this case the compiler will generate the a statement with the correct deployment component location for the newly created object through a statement **new** CWorker(null, dc, ...).

7.1.2 Asynchronous Communication

A very important focus of the code generation is translating the "!" operator. In ABS its syntax is $AsyncCall ::= PureExp\ !\ SimpleIdentifier(PureExp, PureExp)$ and the semantics is to create a new task in the COG that contains the target object identified by $PureExp$. This means that the caller task proceeds independently and in parallel with the callee task, without waiting for the result. The result of evaluating an asynchronous method call expression (Listing 7.3) is a future of type (**Fut**<T>), where T is the return type of the callee method m.

Listing 7.3: Asynchronous method call in ABS

```
1 Fut<Int> f = o!m(e);
```

This type of invocation needs to be wrapped as a lambda expression and sent as a message. The problem here is that lambda expressions only take read-only parameters inside the functions, so all any variables passed as function arguments need to be redeclared before creating the lambda expression. In Java this restriction is defined in terms of the final predicate.

Listing 7.4: Asynchronous method call code generations

```
1 final Integer e1 = e;
2 msg: Callable[ABSFuture[Integer]] = ( )=>o.m(e1);
3 f:ABSFuture[Integer] = o.send(msg);
```

To have the correct code generation of **Fut**<Int> f = o!m(e), where e is a variable of type `Int`, into the code in Listing 7.4 we need to take the following steps:

1. Redeclare all variables that need to be passed to the lambda expression.
2. Create the lambda expression as either a Runnable or Callable.
3. Call the `send` method of the actor to which the asynchronous message is being sent and pass the lambda expression as the argument.

Duplicate Variable Names and Renaming When pre-processing continuations the starting point can be within any scope depth of the method, thus becoming the outer scope of the method wrapping the continuation. The problem that arises here is that the variables declared in this inner scope in the initial code are considered "dead" once the scope ends, therefore their names may be reused. The same variables however are now in the most outer scope of the method that represents the continuation, resulting in duplicate names and variable re-declaration once the names are reused. Also when sweeping the parameters, it is possible that a parameter that is passed to a continuation signature may be re-declared with the same name inside the continuation (i.e. in the case of preprocessing a while block. With all these considerations we need a pattern to rename variables to avoid these name collisions.

7.2 Translation of ABS to Scala

This section covers the biggest challenges that were encountered translating the ABS Standard Library when developing the compiler with the Scala backend support. The standard library is actually a good benchmark for the functional paradigm of ABS as many of the predefined data types are ADTs and many of the defined functions use case expressions and pattern matching. Another topic covered in this section the efficient translation of built-in data types that are dependent on the backend in which they are translated. The challenge for the Scala backend is to find as many corresponding types in Scala that have they same semantic behavior that is specified in the ABS manual. The same goes for built-in functions where ABS imposes backend specific semantics for which it is also better to identify an existing scala function (either in the standard library or otherwise), rather than defining a specific one for the backend.

7.2.1 Sweeping Parameters

In Scala the parameters passed to a method are immutable (val) and because a continuation may change swept parameters we create new local variables to save each parameter. To identify a temporary state from which a message needs to resume, we have to *SWEEP*all of the variables that live in that scope. There are two parts to the sweeping process:

1. During pre-processing, we have to sweep all the variables for their types that live in that scope in order to create the correct method signature.
2. During code generation, we have to sweep for all variables as the pairs of type:value that live in the scope such that we can make the correct method call to be execute when the message is released.

The sweeping process is presented in Figure 7.1. We consider that when *SWEEP* is inside a method definition, we assume it returns tuples of type: value, but when it is inside a method call we assume that it only returns value.

$$
\begin{aligned}
SWEEP(T\ m(\overline{p})\{S\}) &::= \overline{p} \cup SWEEP(\underline{S}) \\
SWEEP(\underline{S_1}; S_2) &::= SWEEP(\underline{S_1}) \\
SWEEP(S_1; \underline{S_2}) &::= SWEEP(\underline{S_2}) \\
SWEEP(case\ e1\ \{P_1 => \{\underline{S_1}\}\}) &::= PSP(P_1),\ await\ or\ get\ statement\ in\ S_1 \\
SWEEP(T\ v_1[= exp]) &::= (type : T,\ value : v_1) \\
SWEEP(\_) &::= \{\emptyset\} \\
PSP(Literal) &::= \{\emptyset\} \\
PSP(QU) &::= \{\emptyset\} \\
PSP(Var) &::= (type : \_inferred,\ value : var) \\
PSP(QU(\overline{p})) &::= \{PSP(a) | a \in \overline{p}\} \\
PSP(\_) &::= \{\emptyset\}
\end{aligned}
$$

Figure 7.1: Parameter Sweeping for a continuation triggered when encountering a given **await**

7.2.2 Compiling Algebraic Data Types and Pattern Matching

These particular features of ABS are the main reason behind using Scala as the backend for ABS rather than Java. Scala provides in the core language constructs the means for translation support to the functional programming paradigm described in Section 3.2. For the translation of an ADT declaration, the example of the `Maybe` data type is very useful **data** Maybe<A> = Nothing | Just(A fromJust);. This examples covers both types of ADT declarations: simple data types and data constructor types. This declaration also incorporates a generic parameter that needs to be correctly translated. The snippet for the compiled Scala code is given in Listing 7.5. The defined type itself (`Maybe`) must be an abstract class such that it cannot be instantiated and only serve as a reference type.

Listing 7.5: Translation of the Maybe ADT

```
1 abstract class Maybe[A<%Ordered[_ >: A]] extends Ordered[Maybe[A]] {
2   var rank : Int;
3 }
```

Like ABS objects, ADTs also have to be comparable using a simple rule that orders the defined types based on the order in which they appear in the declaration. An important observation to make here is that ADTs do not have a superclass and unlike objects ABS does not force different ADTs to be comparable. However, a `rank` (line 2) field needs to be defined to allow comparison between the types defined for an ADTs based on their declaration order. A big advantage of using Scala is that the compiler forces the instantiation of this field in all subclasses and as such can yield an error if anything went wrong in the code generation of the declared ADT types. Coming back to this example the types are in the following relation: $Nothing < Just(...)$. If two ADTs are of the same simple type, they are obviously equal, while two ADTs that are of the same constructor type are compared recursively based on their constructor arguments in a depth-first manner. With these aspects in mind, Listing 7.6 gives an overview of the translation of the two types that the `Maybe` ADT has.

The two types are declared as sub case classes of the `Maybe` class such that they can be referenced correctly. The benefit of using case classes is that they provide a default equals method and as such the pattern matching feature of ABS can have a correspondent to the `match` construct in Scala. On lines 2 and 15 the `rank` variable is initialized with respect to the order of the declarations in the ABS code. In the case of the data constructor type `Just` the `compare` method has to compare the arguments for a proper ordering of two `Just` types (line 20).

7.2.3 Translating the Built-in Types of ABS

In the ABS standard library there are several data types and functions that only have a particular semantics attached to them, without any definition or specification. The general purpose of the Scala backend is to provide an efficient execution platform for ABS, and as such these built-in types must not be redefined in Scala with the expected semantics, but rather already have an existing correspondent type. These are the built-in data types and they are defined in Table 7.1 taken from the ABS Manual. This table is updated with the corresponding type in Scala for a straightforward and efficient translation.

Rational Numbers There are two exceptions to these Scala corresponding types. The first one is for futures which have a much more extensive behavior and semantics in ABS and for this purpose the corresponding type is the `Future` class that is part of the ABS runtime in Java. The second problematic type is the `Rational` type which does not have any pre-defined class in Scala and requires definition of the type in the ABS runtime. A layout of this class is presented in Listing 7.7. The first challenge with this class is that although a rational number may be small, it may be represented by a numerator and denominator that exceed the size `Int` type in Scala. Added to this is that any mathematical operation

Listing 7.6: Translation of the Nothing and Just Types

```
1  case class Nothing[A<%Ordered[_ >: A]]( ) extends Maybe[A] {
2   final var rank : Int = 0;
3
4   def compare( that : Maybe[A]): Int= {
5    if(this.rank == that.rank) {
6     return 0;
7    }
8    else {
9     return this.rank-that.rank;
10   }
11  }
12 }
13
14 case class Just[A<%Ordered[_ >: A]](var fromJust : A) extends Maybe[A] {
15  final var rank : Int = 1;
16
17  def compare( that : Maybe[A]): Int= {
18   if(this.rank == that.rank) {
19    var jthat:Just[A] = that.asInstanceOf[Just[A]]
20    return fromJust.compare(jthat.fromJust)
21   }
22   else {
23    return this.rank-that.rank;
24   }
25  }
26 }
```

Table 7.1: ABS Built-in data types

Name	Description	Example	Corresponding Type in Scala
Unit	The empty (void) type	Unit	Unit
Bool	Boolean values	True, False	Boolean
Int	Integers of arbitrary size	0, -15	Int
Rat	Rational numbers	1/5, 22/58775	-
String	Strings	"Hello World"	String
Fut <A>	Futures	the return type of an asynchronous call	Future< A >
Float	Floating point numbers	3.14, -100.5	Float

may also cause the resulting representation to overflow before it is simplified. For these reasons, both the numerator and denominator are defined as `BigInt` (lines 5 and 6) to preserve correctness of the compiled model. However this adds a significant performance penalty to programs using a large number of rationals, and it is strongly recommended whenever possible to use the newly introduced ABS built-in type `Float` which has a corresponding primitive in Scala with the same name. This is especially important for models where high performance is required.

Listing 7.7: The Rational Class in the ABS Runtime

```
1  class Rational(n: BigInt, d: BigInt) extends Ordered[Rational] {
2    require( d != 0 )
3
4    private val g = Rational.gcd(n.abs, d.abs)
5    val numer: BigInt = n/g * d.signum
6    val denom: BigInt = d.abs/g
7
8    def this(n: Int) = this(n, 1)
9    def this(rational: Rational)= this(rational.numer,rational.denom);
10
11   override def toString = ""+ numer + (if (denom == 1) "" else ("/"+denom))
12
13   // default methods (for all math operations)
14   def +(that: Rational): Rational =
15     new Rational( number * that.denom + that.number * denom,
16         denom * that.denom )
17   def +(that: Int): Rational =
18     new Rational( numer + (that * denom), denom )
19
20   ...
21 }
```

Another challenge specific to the Rational type is that, because it does not have a pre-defined type in Scala, it cannot be used together with other numerical types like the semantics of ABS require. As such the Java ABS runtime has to define this behaviour for mathematical operations (line 18) and comparison operations as shown in Listing 7.8. In these examples the methods are defined for operations between rationals and integers and the same behaviour is defined for operations between rationals and floats.

Finally there will be certain instances where an integer will have to be "promoted" to a rational such as on an assignment statement Rational x = 0;. The Rational object in Listing 7.9 defines in Scala

Listing 7.8: The Comparison Methods in the Rational Class

```
1  def equals(that: Rational) = {
2    (this.numer==0 && that.numer==0)||
3    (this.numer == that.numer && this.denom == that.denom)
4  }
5
6  def equals(that: Int) = {
7    (this.numer == that && this.denom == 1) || (this.numer == 0 && that == 0)
8  }
9
10 override def equals(obj: scala.Any): Boolean = {
11   if(obj.isInstanceOf[Int]){
12     return equals(obj.asInstanceOf[Int])
13   }
14   if(obj.isInstanceOf[Rational]){
15     return equals(obj.asInstanceOf[Rational])
16   }
17   return super.equals(obj)
18 }
19
20 override def compare(that: Rational) = {
21   val c = this.numer * that.denom - that.numer * this.denom
22   if(c.isValidInt)
23     c.toInt
24   else if(c<0)
25     Int.MinValue
26   else
27     Int.MaxValue
28 }
```

this implicit conversion (line 2), together with a constructor that creates a rational from an integer parameter(line 6). Finally in order to reduce a rational number to its simplest form and also to avoid storing too many `BigInt` numbers the Rational object provides a function for calculating the greatest common divisor (gcd on line 3) to be applied straight after constructing the number. Another performance issue that appears when using Rational numbers is that they are immutable, but they are not Scala primitives so a large number of these objects are created when a lot of computation is performed with them. As they are no pools of constants defined like String and Integer pools it is best to avoid using this built-in type in favor of Float to model high-performance applications.

Listing 7.9: The Rational Object in the ABS Runtime in Java

```
1 object Rational {
2   implicit def intToRational(x: Int) = new Rational(x)
3   private def gcd(a: BigInt, b: BigInt) : BigInt = if (b == 0) a else gcd(b, a % b)
4
5   def apply(numer: Int, denom: Int) = new Rational(numer, denom)
6   def apply(numer: Int) = new Rational(numer)
7 }
```

7.3 Compiler Correctness

Implementing cooperative scheduling in ABS that scales in the number of active objects and messages provides a major challenge (as described in Chapter 4). Furthermore, it also complicates considerably reasoning about the correctness of ABS programs. This is evidenced by that no sound and complete proof system for ABS has been introduced yet. The proof system in [DO14] for example is shown to be complete only for the sublanguage that does not support cooperative scheduling. ABS itself has a formal semantics, but the languages to which it is compiled, Java and Scala, do not have a full language formal semantics. Therefore it is difficult to provide a formal correctness proof for the whole compiler and all the challenges described in this chapter. As such, the focus is on investigating correctness of certain aspects of the compiler, mainly the translation of the expressive power of cooperative scheduling in ABS. This section shows that the powerful abstraction of cooperative scheduling in ABS can be modeled by a run to completion model of active objects.

The essence of this compilation process is the spawning mechanism the translation is generalized to a run to completion model described by a language GAC (Guarded ACtor) which provides a guarded-command language ([Dij78]) for the description of the method bodies. The formal translation of ABS into GAC is given by an intermediate language ABS-SPAWN which uses an explicit spawn operation to model the execution of await statements. In the GAC language the operation of spawning local processes can be modeled directly by asynchronous self-calls. Any source code written in ABS consists of a set of classes, and each class consists of a set of method definitions. In this section we abstract from the nominal type system of ABS and its functional layer, and focus on the control flow of ABS programs. Figure 7.2 presents the formal syntax of ABS statements which are used to describe the method bodies.

The expression *e* denotes a local side-effect free expression (that is, its evaluation only depends on the local state of the actor and does not affect this local state). For the purpose of providing a smooth and clear proof, we can abstract from its syntax (which in general involves the functional layer of ABS). For notational convenience we assume that every method call (asynchronous or synchronous self call) returns a value. We assume these values typed according to the type system of ABS. A slight modification of Listing 3.8 is shown is Listing 7.10 where it is imposed in the formal syntax that returned expressions are side-effect free. As such, the return instructions is separated into the instructions on lines 10 and !11.

Further we restrict a guard g of an **await** statement by either a local side-effect free Boolean condition *b* or a single a future variable. It is not difficult to see that this restriction does not restrict the expressive

S	::=	ϵ	empty statement
	|	$x = e$	basic assignment
	|	$y = x!m(\bar{e})$	asynchronous method call
	|	$y = m(\bar{e})$	synchronous method call
	|	$x =$ **new** $C(\bar{e})$	object creation
	|	**await** g	await statement
	|	$x = y$.**get**	get statement
	|	**if** b $\{S\}$ **else** $\{S\}$	conditional statement
	|	$S; S$	sequential composition
	|	**return** e	return statement

Figure 7.2: Syntax for ABS statements.

Listing 7.10: Synchronous Call in ABS

```
1  Worker getWorker( ){
2    await !(emptySet(workers));
3    Worker w = take(workers);
4  }
5
6  Result sendWork( ) {
7    Worker w = this.getWorker( );
8    workers = remove(workers, w);
9    Fut<Result> f = w ! doWork( );
10   Result r = f.get;
11   return r;
12 }
```

power since any **await** statement on a guard which consists of a Boolean condition and a set $\{x_1, \ldots, x_n\}$ of futures can be implemented by a sequential composition:

```
1  await x1?; ...; await xn?; await b;
```

because a future is single-write shared data (note that the **await** on the Boolean condition should indeed be executed last).

For technical convenience, we also abstract from the so-called Concurrent Object Groups (COG) as provided by the ABS language. However, it is not difficult to generalize the main result to the language including COG's. More importantly, it should be noted that the syntax does not include the usual **while** statement. A detailed discussion of the challenges of translating **await** statements occurring in the body of a **while** statement will be presented in section 7.3.2. Note however that the **while** statement can be modeled by tail recursion, using synchronous self calls.

To allow for this tail-recursion we assume for the ABS language to feature synchronous self calls of the form $m(\bar{e})$.

7.3.1 ABS Operational Semantics

This section presents a different approach to the semantics of the ABS language using variable renaming of local variables instead of local environments. This allows for a simple definition of a process as the statement to be executed, which in turn allows for a transparent way of modeling cooperative scheduling.

We assume given an ABS program P where object configurations are of the form (σ, S, Q):

- σ assigns values to the instance variables (fields) of the class (we treat the keyword **this** as a distinguished instance variable identifying the object) and all the fresh variables generated for the local variables of the different method invocations. For any side-effect free expression e (including Boolean conditions b) we denote by $\sigma(e)$ the value of e in σ.
- S represents the current statement of the active process that is run by the actor denoted by $\sigma(\mathbf{this})$.
- Q is a (multi-)set of statements which represent suspended processes.

We define a global configuration G as a pair (F, O) where F is a partial function which assigns to each future identity f in its domain a value $F(f)$ and O is a set of configurations (as defined above). By $F(f) = \perp$ we denote that the future f has not been completed yet. For handling the completion of futures by return statements, we introduce an implicit formal parameter **dest** which holds the value returned by the method invocation. In the rules below we assume some mechanism for generating fresh variables. Generating fresh variables is needed to distinguish between the **dest** variable of each method and also to avoid name clashes when renaming local variables. To make the use of the **dest** variable explicit, we replace every **return** statement in a method body with the auxiliary statement **return** *e to* **dest**.

The following rule describes the operation semantics of an assignment.

Assignment Rule

$$(F, \{(\sigma, x = e; S, Q)\} \cup O) \rightarrow (F, \{(\sigma[x = \sigma(e)], S, Q)\} \cup O)$$

Here and in the sequel we denote by $\sigma[x = v]$ the update of σ which assigns the value v to the variable x.

Asynchronous Invocation Rule The following rule describes the semantics of an asynchronous method call.

$$(F, \{(\sigma, y = x!m(\bar{e}); S, Q), (\sigma', S', Q')\} \cup O)$$
$$\rightarrow$$
$$(F[f = \perp], \{(\sigma[y = f], S, Q), (\sigma'', S', Q'')\} \cup O)$$

where:

- f is a new future which does not exist in the domain of F (and thus $F[f = \perp]$ denotes the function which results from extending the domain of F by assigning $\perp$ to f)
- $\sigma(x) = \sigma'(\mathbf{this})$.
- S' is the current statement of the active process run by the target object x (the callee in $x!m(\bar{e})$)
- Q'' extends Q' with the body of method m where all the formal parameters (including the distinguished variable **dest**) are replaced by fresh (that is, not in use in (σ', S', Q')) variables.
- σ'' results from assigning the values of the actual parameters $\sigma(\bar{e})$ to the corresponding fresh local variables. Additionally $\sigma''(\mathbf{dest}') = f$ where $\mathbf{dest}'$ is the fresh local variable corresponding to the destiny variable **dest**.

Synchronous Self Call Rule

$$(F, \{(\sigma, x = m(\bar{e}); S, Q)\} \cup O) \rightarrow$$
$$(F[f = \perp], \{(\sigma', S'; x = f.\mathbf{get}; S, Q)\} \cup O)$$

where:

- f is a new future which does not exist in the domain of F (and thus $F[f = \perp]$ denotes the function which results from extending the domain of F by assigning $\perp$ to f).
- f is the future that will hold the result of the asynchronous method invocation m.
- S' is obtained by renaming the local variables in the body of method m (as above, including the variable **dest**) by fresh variables and σ' assigns to these fresh variables the values of the actual parameters $\sigma(\bar{e})$. Additionally $\sigma'(\mathbf{dest'}) = f$ where $\mathbf{dest'}$ is the fresh local variable corresponding to the destiny variable **dest**. This translation of the body replaces the return statement with an auxiliary statement **return** *e to* **dest**. Freshness is defined as a variable not in use by any statement in S' and Q' .

Note that we thus use simple inlining which works because we introduce fresh variables for the formal parameters of methods. We use a future in order to get a uniform semantics for returning a value for both synchronous calls and asynchronous calls. The get operation will always be enabled (because of the assumption that any method body will end with a return statement), but it is used here instead of an await because we want the process to proceed, as otherwise we would have a release point which breaks the call stack.

In the case of a method which is declared void then the syntax would be **return null** *to* **dest**. By means of this convention, every suspended statement is uniquely identified by its destiny variable,so that we can model Q as a set..

Object Instantiation Rule

$$(F, \{(\sigma, x \ = \ \mathbf{new}\ C(\bar{e}); S, Q)\} \cup O) \rightarrow$$

$$(F, \{(\sigma[x = o], S, Q), (\sigma', S', \emptyset)\} \cup O)$$

where:

- o is a fresh object identity (not appearing as value of a variable in the initial configuration).
- S' is the constructor method body.
- $\sigma'(this) = o$ and σ' assigns to the formal parameters of the constructor method the values $\sigma(\bar{e})$.

Note that each object configuration assigns a new object identity to the instance variable $this$. This explains the usage of union in object configurations.

Conditional Statement Rule The conditional statement has the following two rules.

$$(F, \{(\sigma, \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2; S, Q)\} \cup O) \rightarrow$$

$$(F, \{(\sigma, S_1; S, Q)\} \cup O)$$

where $\sigma(b) = true$.

$$(F, \{(\sigma, \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2; S, Q)\} \cup O) \rightarrow$$

$$(F, \{(\sigma, S_2; S, Q)\} \cup O)$$

where $\sigma(b) = false$.

Return Rule

$$(F, \{\sigma, \textbf{return } e \ \ to \ \textbf{dest}', Q)\} \cup O) \rightarrow (F[f = \sigma(e)], \{(\sigma, \epsilon, Q)\} \cup O)$$

where $f = \sigma(\textbf{dest}')$

Get Rule

$$(F, \{(\sigma, x = y.\textbf{get}; S, Q)\} \cup O) \rightarrow (F, \{(\sigma[x = F(\sigma(y))], S, Q)\} \cup O)$$

where $F(\sigma(y)) \neq \perp$.

Await Rule

$$(F, \{(\sigma, \textbf{await } g; S, Q)\} \cup O) \rightarrow (F, \{(\sigma, \epsilon, \{\textbf{await } g; S\} \cup Q)\} \cup O)$$

where ϵ represents the empty statement, denoting that the current executing statement has ended. This rule "blindly" suspends the current statement without evaluating the guard. The evaluation of the guards will be performed in the context of the scheduling rule below.

Scheduling Rule The following rules schedule enabled **await** statements of a Boolean and a future variables respectively. For suspended statements that start with an **await** we have the following two rules.

$$(F, \{(\sigma, \epsilon, \{\textbf{await } b; S\} \cup Q)\} \cup O) \rightarrow (F, \{(\sigma, S, Q)\} \cup O)$$

where $\sigma(b) = true$

$$(F, \{(\sigma, \epsilon, \{\textbf{await } y; S\} \cup Q)\} \cup O) \rightarrow (F, \{(\sigma, S, Q)\} \cup O)$$

where y is a future variable such that $F(\sigma(y)) \neq \perp$.

For any other suspended statement that is in Q, e.g., that resulted from an asynchronous call, we have the following rule:

$$(F, \{(\sigma, \epsilon, \{S\} \cup Q)\} \cup O) \rightarrow (F, \{(\sigma, S, Q)\} \cup O)$$

7.3.2 ABS-SPAWN

In this section we introduce the ABS-SPAWN language which is obtained from the ABS language discussed above by replacing the await statement with a statement **spawn**(g, S), the so-called **spawn** statement, for spawning a new local process that executes the statement S. In ABS-SPAWN method invocations are thus executed in a run-to-completion mode.

For the operational semantics of the ABS-SPAWN language we introduce the run-time syntax construct $(g \rightarrow S)$ that represents a suspended statement (S) that is guarded by an enabling condition (g). We thus make a distinction between the statement that spawns a process and resulting generated suspended process. The guard in $(g \rightarrow S)$ is the enabling condition for scheduling the corresponding statement S for execution. For its semantics we used the same notions for a global configuration and object configuration, as introduced for the semantics of the ABS language, The semantics of the ABS-SPAWN language results from the semantics of ABS by replacing the **Await Rule** with the rule **Spawning Subtasks** and changing the **Scheduling Rule**, as described above.

Spawning Subtasks Spawning a sub-task simply consists of adding a corresponding statement with an enabling condition to the set Q of suspended processes:

$$(F, \{(\sigma, \textbf{spawn}(g, S); S', Q)\} \cup O) \rightarrow (F, \{(\sigma, S', \{(g \rightarrow S)\} \cup Q)\} \cup O)$$

Scheduling Rule The following rules describe the scheduling of an enabled suspended task. The first two rules are for statement suspended by **await**.

$$(F, \{(\sigma, \epsilon, \{(b \rightarrow S)\} \cup Q)\} \cup O) \rightarrow (F, \{(\sigma, S, Q)\} \cup O)$$

where $\sigma(b) = true$.

$$(F, \{(\sigma, \epsilon, \{(y \rightarrow S)\} \cup Q)\} \cup O) \rightarrow (F, \{(\sigma, S, Q)\} \cup O)$$

where y is a future variable and $F(\sigma(y)) \neq \bot$.

The last rule is for statements that are suspended as a result of an asynchronous invocation and is the same as in the ABS operational semantics:

$$(F, \{(\sigma, \epsilon, \{S\} \cup Q)\} \cup O) \rightarrow (F, \{(\sigma, S, Q)\} \cup O)$$

Translating ABS into ABS-SPAWN We next introduce a formal translation from ABS programs into ABS-SPAWN programs. This translation is applied to every class in the ABS program. For each class, every method body is viewed as a sequential composition of the first instruction followed by its (sequential) continuation and translated accordingly.

$$\begin{aligned}
T(\epsilon) &:= \epsilon \\
T(x = e;\ S) &:= x = e; T(S) \\
T(\textbf{await}\ g;\ S) &:= \textbf{spawn}(g,\ T(S)\ ) \\
T(\textbf{if}\ b\ \{S_1\}\ \textbf{else}\ \{S_2\}; S) &:= \textbf{if}\ b\ \{T(S_1;\ S)\}\ \textbf{else}\ \{T(S_2;\ S)\} \\
T(y\ =\ x!m(\bar{e});\ S) &:= y = x!m(\bar{e});\ T(S) \\
T(y\ =\ m(\bar{e});\ S) &:= y\ =\ m(\bar{e});\ T(S) \\
T(x\ =\ \textbf{new}\ C(\bar{e}); S) &:= x = \textbf{new}\ C(\bar{e});\ T(S) \\
T(x = y.\textbf{get};\ S) &:= x = y.\textbf{get};\ T(S) \\
T(\textbf{return}\ e;\ S) &:= \textbf{return}\ e;\ T(S)
\end{aligned}$$

Figure 7.3: Translation of ABS into ABS-SPAWN

The scheme is applied using a bottom-up approach starting at the level of statements using Figure 7.3. The scheme is then lifted to the level of method bodies. Finally a translation of a class simply consists of the translation of its method definitions.

In Figure 7.3 the empty statement is denoted by ϵ (we assume here the syntactical equivalence $S; \epsilon \equiv S$). The translation of an await construct with guard g followed by a (sequential) continuation S results simply in a spawn statement with two parameters: the guard g and the task representing the translation applied to the continuation ($T(S)$). A conditional statement is translated by "absorbing" the sequential continuation that follows into the two branches of the statement. This general pattern also would apply to, for example, the translation of the ABS case statement (or pattern matching statement) where the continuation has to capture for each possible pattern (P_i) both the block to be executed on that pattern branch (S_i) as well as the rest of the control flow that follows the statement (S). The translation thus captures the whole syntactic continuation that follows an await statement as the new task to be spawned. Therefore the translation of the method containing the await statement will terminate directly after having spawned the corresponding subtask, thus emulating an implicit suspension point.

The While Statement. We describe next the problem of translating a repetitive loop or the while statement. Intuitively, to capture the syntactic continuation that follows an await statement occurring in the body of the while statement, the translation could simply “unfold" the loop. However this would result in a recursive translation. Instead, we can model while statements by means of a tail recursive method. Note that such a method should capture in its formal parameters the execution context (that is, all the local variables used in the loop body).

Listing 7.11: While Loop in ABS

```
1  {  List<Fut<Int>> futuresList = Nil;
2
3   //ABS code that fills the futuresList with
4   //futures resulting from asynchronous calls
5
6   this.sum=0;
7   while( !emptyList( futuresList ) ){
8    Fut<Int> f = head( futuresList );
9    await f?;
10   Int x = f.get;
11   this.sum = this.sum + x;
12   futuresList = tail( futuresList );
13  }
14  if( this.sum > 0 ){
15   //do work
16  }
17 }
```

To describe this in more detail, we look at an example in Listing 7.11 that computes the sum of numbers generated by asynchronous calls whose results are captured in a list of futures. In ABS, lists are part of the functional layer and all functions applied on them (`head, tail, emptyList`) are side-effect free. We note that in this particular program the variables `x, f` are local variables declared inside the repetitive loop, `futuresList` is a local variable defined in the method’s body prior to the loop scope and `sum` is a class member variable.

This repetitive loop can naturally be “unfolded" by defining a new method `m` with a formal parameter of type `List< Fut<Int»`, as we observe it is the only local variable declared prior to the loop. This is shown in Listing 7.12. We can see that this way of “unfolding"" the loop works because the state of execution (in this case, the continuously processed list) is passed to the next call as formal parameters. Listing 7.13 then shows the translation of tail-recursive method modeling the while statement. Note that the syntactic continuation of the await statement is captured in this translation by the recursive call.

Listing 7.13: Translation Tail Recursion

```
1  Unit m(List<Fut<Int>> futuresList){
2   if(!emptyList(futuresList)){
3    Fut<Int> f = head(futuresList);
4    spawn( f?, {
5     Int x = f.get;
6     this.sum = this.sum + x;
7     futuresList = tail(futuresList);
8     m(futuresList)
9    } );
10  }
11 }
```

Listing 7.12: Re-written While Loop in ABS using tail recursion

```
1  //new method
2  Unit m(List<Fut<Int>> futuresList){
3   if(!emptyList(futuresList)){
4    Fut<Int> f = head(futuresList);
5    await f?;
6    Int x = f.get;
7    this.sum = this.sum + x;
8    futuresList = tail(futuresList);
9    m(futuresList);
10  }
11 }
12
13 {  //original method scope
14  this.sum=0;
15  m(futuresList);
16  if(this.sum > 0){
17   //do work
18  }
19 }
```

To conclude the presentation of ABS-SPAWN, we apply the translation scheme to the `WorkerPool` class written in ABS in Listing 3.6. The resulting code in ABS-SPAWN is illustrated in Listing 7.14.

Listing 7.14: The Worker Pool Class in ABS-SPAWN

```
1  class WorkerPool(){
2   Set<Worker> workers;
3
4   Result sendWork() {
5    spawn ( !( emptySet(workers) ) , {
6     Worker w = take(workers);
7     workers = remove(workers, w);
8     Fut<Result> f = w ! doWork();
9     spawn (f?, {
10     Result result = f.get;
11     return result;
12    } ) ;
13   } );
14  }
15
16  Unit finished(Worker w) {
17   workers = insertElement(workers, w);
18  }
19 }
```

Correctness of the ABS Translation In order to show the correctness of the above translation of ABS programs into ABS-SPAWN programs, we use G to denote a global ABS configuration as well as

ABS-SPAWN configurations. We introduce the notation:

$$G \rightarrow_{\text{abs}} G'$$

to differentiate between transitions in pure ABS and transitions in ABS-SPAWN which are denoted as:

$$G \rightarrow G'$$

Let $T(G)$, for any global ABS configuration G, denote the result of applying the translation to all the *executing* ABS statements in G and translating any *suspended* **await** statement **await** $g; S$ in G by $g \rightarrow T(S)$. We now can state the following theorem which states the correctness of the translation of **await** statements in ABS, the proof of which proceeds by a straightforward case analysis of the first instruction of an executing statement.

Theorem 1. *For any configuration G of an ABS program we have:*

$$G \rightarrow_{abs} G' \textit{ iff } T(G) \rightarrow T(G')$$

Proof. The proof proceeds by a case analysis of the transition rules. We treat the following main cases. We only need consider those statements that are affected by the translation, because for statements like the *assignment* and *empty* statement, the semantics of ABS coincides with that of ABS-SPAWN. We only consider the main case of translating the **await** statement, because the translation of the conditional statement is correct because of standard programming equivalences.

Figure 7.4: Execution of an Await Statement

$$\begin{array}{ccc} \Sigma[(\sigma, \textbf{await } g;\ S, Q)] & \xrightarrow[\text{ABS}]{} & \Sigma[(\sigma, \epsilon, \{\textbf{await } g;\ S\} \uplus Q)] \\ \Big\downarrow T & & \Big\downarrow T \\ T(\Sigma)[(\sigma, \textbf{spawn}(\ g;\ T(S)), T(Q))] & \xrightarrow[\text{ABS-SPAWN}]{} & T(\Sigma)[(\sigma, \{g \rightarrow T(S)\} \uplus T(Q))] \end{array}$$

The proof is divided into two parts. The first part is presented by the diagram in Figure 7.4 and treats the translation of the **await** statement that appears as an instruction in a context Σ, that is, $\Sigma[(\sigma, S, Q)]$ describes a global configuration (F, O), with $(\sigma, S, Q) \in O$. The upper transition corresponds to the application of the **Await Rule**, the result of which, namely that the process **await** $g;\ S$ is added to the suspended processes Q (of the executing active object), is denoted by the corresponding global configuration $\Sigma[(\sigma, \epsilon, \{\textbf{await } g;\ S\} \uplus Q)]$. The lower transition results from the definition of the translation scheme to global configurations, and a corresponding application of the **Spawning Tasks** rule in ABS-SPAWN.

Figure 7.5: Scheduling a Suspended Statement

$$\begin{array}{ccc} \Sigma[(\sigma, \epsilon, \{\textbf{await } g;\ S\} \uplus Q)] & \xrightarrow[\text{ABS}]{} & \Sigma[(\sigma, S, Q)] \\ \Big\downarrow T & & \Big\downarrow T \\ T(\Sigma)[(\sigma, \epsilon, \{g \rightarrow T(S)\} \uplus T(Q))] & \xrightarrow[\text{ABS-SPAWN}]{} & T(\Sigma)[(\sigma, T(S), T(Q))] \end{array}$$

Conversely, the second part is presented by the diagram in Figure 7.5 and shows the correctness of translating the **await** statement as part of a suspended process which conforms to the semantics of the **Scheduling Rules** in ABS and ABS-SPAWN, respectively.

7.4 The GAC Language

It is worthwhile to note that by the above translation scheme we can actually embed ABS in a language *without* any await statements (that allow for cooperative scheduling) by encoding **spawn**(g, S) itself as an asynchronous self call of the form **this**$!m()$, where $m()$ is an unique method name with defining body $g \to S$.

In Figure 7.6 we introduce so-called guarded command statements (following [Dij78]) as statements for describing the method bodies in ABS.

S	$::=$	ϵ	empty statement
	$\mid$	$x = e$	basic assignment
	$\mid$	$y = x!m(\bar{e})$	asynchronous method call
	$\mid$	$y = m(\bar{e})$	synchronous method call
	$\mid$	$x =$ **new** $C(\bar{e})$	object creation
	$\mid$	$x = y.$**get**	get statement
	$\mid$	$(*)\Box_{i=1}^{n}\ g_i \to \{S_i\}$	guarded command
	$\mid$	**case** $e\ \{e \Rightarrow S\}$	case statement
	$\mid$	$S; S$	sequential composition
	$\mid$	**return** e	return statement

Figure 7.6: ABS guarded command statements.

The semantics of the statement $\Box_{i=1}^{n}\ g_i \to S_i$ consists of a non-deterministic selection of one of the statements S_i for which the associated guard g_i is enabled. It blocks the execution of the active object if none of the guards are enabled. A guard itself in the GAC language consists of a Boolean condition and a set of futures. Such a guard is enabled if the Boolean condition holds and all its futures are completed (that is, for all of them the return value has been produced). Its iterated version (indicated by the asterisk) consists of repeatedly executing the marked guarded choice as long as one of its guards is enabled. It terminates as soon as none of the guards is enabled. Formally, the semantics of the guarded command statements is described by the following rules (the semantics of the other statements are described as in the ABS semantics).

Guarded Choice Rule

$$(F, \{(\sigma, \Box_{i=1}^{n}\ g_i \to \{S_i\}; S, Q)\} \cup O) \to (F, \{(\sigma, S_j; S, Q)\} \cup O)$$

provided g_j is enabled in σ and F.

For its iterated version we have the following two transitions.

Iterated guarded Choice Rule

$$(F, \{(\sigma, * \Box_{i=1}^{n}\ g_i \to \{S_i\}; S, Q)\} \cup O) \to$$

$$(F, \{(\sigma, S_j; * \Box_{i=1}^{n}\ g_i \to \{S_i\}; S, Q)\} \cup O)$$

provided g_j is enabled in σ.

$$(F, \{(\sigma, * \Box_{i=1}^{n}\ g_i \rightarrow \{S_i\}; S, Q)\} \cup O) \rightarrow (F, \{(\sigma, S, Q)\} \cup O)$$

provided none of the g_j is enabled in σ and F.

Further, we have the following scheduling rules.

Scheduling Rules

$$(F, \{(\sigma, \epsilon, \{\Box_{i=1}^{n}\ g_i \rightarrow \{S_i\}; S\} \cup Q)\} \cup O) \rightarrow (F, \{(\sigma, S_j; S, Q)\} \cup O)$$

provided g_j is enabled in σ and F.

$$(F, \{(\sigma, \epsilon, \{* \Box_{i=1}^{n}\ g_i \rightarrow \{S_i\}; S\} \cup Q)\} \cup O) \rightarrow$$

$$(F, \{(\sigma, S_j; * \Box_{i=1}^{n}\ g_i \rightarrow \{S_i\}; S, Q)\} \cup O)$$

provided g_j is enabled in σ and F.

The resulting GAC language thus follows a strict run to completion mode of execution of the methods by active objects, like the Rebeca language)[Sir07]). Differently from the Rebeca language, it features guarded command statements which allow to associate an enabling condition with a suspended process. Note that such a suspension mechanism avoids modeling a suspended process $g \rightarrow S$ by a recursive method definition

$$m()\{\textbf{if}\ g\ \{S\}\ \textbf{else}\ \{\textbf{this}!m()\}\}$$

which involves busy waiting (and which assumes testing a future as a Boolean condition).

As an overall conclusion we illustrate in Figure 7.15 the translation of the `WorkerPool` into GAC. Note the need to wrap the statements of the guarded commands into separate methods. Thus these methods can be called asynchronously and stored as suspended messages into the queue of the `WorkerPool` until their guards are enabled. As such, execution of other enabled statements can continue without blocking the actor.

Listing 7.15: The Worker Pool Class in GAC

```
1  class WorkerPool(){
2    Set<Worker> workers;
3
4    Result sendWork() {
5      ! emptySet(workers) → this ! m1( workers );
6    }
7
8    Result m1(Set<Worker> workers){
9      Worker w = take(workers);
10     workers = remove(workers, w);
11     Fut<Result> f = w ! doWork();
12     f → this ! m2( f );
13   }
14
15   Result m2(Fut<Result> f){
16     Result result = f.get;
17     return result;
18   }
19
20   Unit finished(Worker w) {
21     workers = insertElement(workers, w);
22   }
23 }
```