Project N°:	**FP7-610582**
Project Acronym:	**ENVISAGE**
Project Title:	**Engineering Virtualized Services**
Instrument:	**Collaborative Project**
Scheme:	**Information & Communication Technologies**

Deliverable D3.1
Code Generation

Date of document: T24

Start date of the project: **1st October 2013** Duration: **36 months**

Organisation name of lead contractor for this deliverable: **CWI**

Final version

STREP Project supported by the 7th Framework Programme of the EC		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Executive Summary:
Code Generation

This document summarises deliverable D3.1 of project FP7-610582 (Envisage), a Collaborative Project supported by the 7th Framework Programme of the EC within the Information & Communication Technologies scheme. Full information on this project is available online at `http://www.envisage-project.eu`.

D3.1 reports on the results achieved for automatic generation of target code from ENVISAGE's modeling language. The deliverable is a prototype and description of the code generation tool.

The main focus of this deliverable is to present the effort towards making the Abstract Behavioral Specification language an easy to use and flexible language for designing applications that concretely run programs to be used in research and development as well as industry. In this deliverable we want to promote ABS as a programming language in order to observe its behavior when executing on actual hardware. We want to be able to detect issues that are frequent in large-scale parallel and distributed programs such as performance bottlenecks, synchronization issues, resource provisioning and concurrency problems.

The main objective of this task is to generate executable code and test it exactly like applications written in other programming languages to achieve a reliable and efficient end product. We will see how error-prone the generated code is from simple software bugs and runtimes errors to reliability issues of larger applications like resource starvation and hardware failures. We selected the following languages to serve as backends from our code generation tool:

- Java, a very well-known and mainstream language at industry-level for developing large-scale applications.
- Haskell, a functional language that allows a very intuitive and simple mapping of all the ABS features.

List of Authors

Frank de Boer (CWI)
Nikolaos Bezirgiannis (CWI)
Vlad Serbanescu (CWI)
Behrooz Nobakht (CWI)
Elvira Albert (UCM)
Enrique Martin-Martin (UCM)

Contents

Chapter 1

Technical Summary

Deliverable D3.1 documents the design and implementation of two of recent backend developments for Abstract Behavior Intefaces ABS in the context of Envisage project.

ABS can be compiled to a target programming language, referred to as a *backend*. A generic architecture for modelling, implementation and tooling of a backend is created such that:

- it facilitates a *generic common* set of necessary libraries for every backend implementation
- it facilitates *modular development* of components for every backend that is required for verification and quality assurance
- it does *not* impose any specific limitation/requirement for the backend development

The architecture is presented in Figure 1.1:

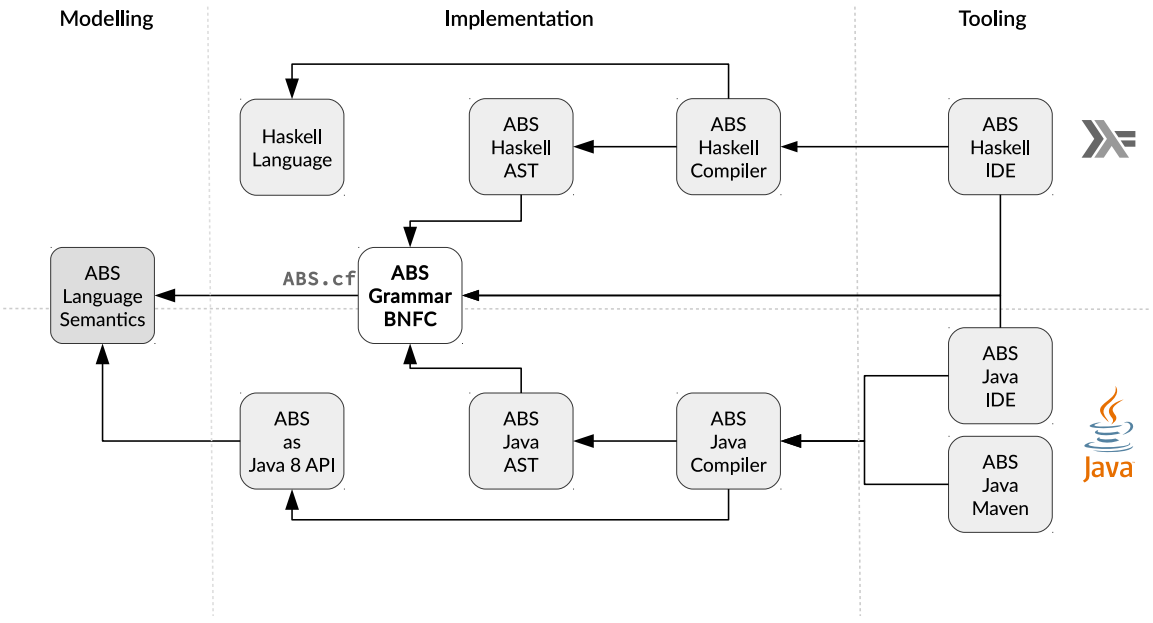

Figure 1.1: Generic architecture for ABS backend development

In Figure 1.1:

- ABS modelling language specifies the language features and provides formal semantics on top of which the development is based.
- `ABS.cf` is the *central* ABS language syntax grammar as an LBNF grammar used by BNFC [1].

[1] `http://bnfc.digitalgrammars.com/`

- *Haskell Language* or *ABS as Java 8 API* present parts of the target language features or runtime that provide semantic mapping to that of the ABS language. This component of the architecture essentially *enables* programmers to use ABS language features, if it fits, *without* the need to first compile ABS to the target language as an initial requirement.
- The *Abstract Syntax Tree* (AST) generated for a particular target language (Java or Haskell) by BNFC. This component of the architecture provides *extension* points such that it facilitates model checking and type checking tools to be able to apply their logic during/orthogonal to the compilation phases.
- *Compiler* (e.g. Haskell or Java) is the compiler component that translates an ABS program into the target language source.
- Tooling and IDE development refers to how different tooling development can depend on the "implementation" layer components of the architecture to provide better support of the ABS language to the backend language programmer.

The main technical contents of D3.1 consists of the four main appendices:

1. The library used in the code generation process of the executable Java code based on paper [9]
2. The cloud computing features of ABS based on paper [6]
3. The formal translation of ABS into Haskell based on paper [4]
4. The benchmark evaluation for all existing backends in ABS.

The rest of the deliverable briefly describes the two backends that generate executable code. The code will be translated in two programming languages that best fit the project overall aim and impact on the industry as well as supporting all the features that our modeling language has. In Chapter 2 we motivate the choice for choosing the Java language and explain the process of translating ABS modeling code into Java executable code. The code generation tool is available online on the github repository at `https://github.com/CrispOSS/jabsc.git`. In Chapter 3 we present the motivation for selecting Haskell and describe the link between each language feature of ABS and its corresponding syntax in Haskell. Chapter 4 presents our effort to provide resource bounds for the compiled programs in the Haskell backend.

One of the goals within the code generation task is to be able to provide also resource bounds for the compiled programs. For this, ideally the upper bounds obtained by the resource analysis of the ABS models should be "transferred" to the compiled code. In order to ensure the soundness of this step, we have formally proved that the transformation preserves the resource consumption. We have done this for the concurrent-object subset of the ABS language and the Haskell compiler, i.e., given a cost model that assigns a cost to each instruction of the ABS language, we prove that the cost of executing the Haskell-translated program is the same as executing the original ABS program since both execute the same instructions. The proof required to make a formal statement of the soundness of the translation of ABS into Haskell which is expressed in terms of a simulation relation between the operational ABS semantics and the semantics of the generated Haskell code. The soundness claim ensures that every Haskell derivation has an equivalent one in the ABS code. However, since for efficiency reasons, the translation fixes a selection order between the objects and the tasks within each object, we do not have a completeness result. Having the soundness result we can easily ensure that the upper bounds on the resource consumption obtained by the resource analysis of the original ABS program are preserved during the Haskell compilation. Thus, they are valid upper bounds for the Haskell-translated program as well.

Chapter 2

Java Backend

2.1 Motivation and Challenges

The Abstract Behavioural Specification (ABS) language [8] offers several layers of modelling parallel and distributed applications with possibilities for static analysis for costs and correctness[3, 5, 2, 7]. One of the most important layers of ABS is its object-oriented model which offers a Java-like syntax enhanced with several new constructs and annotations that allow design of distributed applications for grid and cloud environments. Furthermore, applications can be modeled with custom defined schedulers that model the behaviour needed by the service provider using constructs for asynchronous method calls and preemption. ABS objects are constructed implicitly with scheduling queues with a default first-in-first-out (FIFO) order. Another significant part of ABS is its functional layer that allows modeling of applications using a functional paradigm that supports high stacks of recursion, pattern matching and efficient evaluations of data structures at runtime.

At a conceptual and modeling level the code is very easy to write and analyze it, however from this model we need to generate a program that will execute the distributed application in a cloud environment with the modeled scheduling possibilities and both functional and object-oriented paradigms, as our final goal is to generate valid production code. This means that we need to translate into an object-oriented language (Java) the execution of an object's methods invocations, preemption and process queue, as well as recursive functions, data type definitions, predicates and lambda expressions. To do this, until Java 8, we only had the possibility of modeling parallel execution with Java Threads and this resulted in the creation of a large number for this particular scenario. With the introduction of lambda expressions in Java 8 the only issue that we need to solve is how to efficiently save the call stack when the need for preemption appears without using a memory-expensive Thread. In this section of our deliverable we explain the challenges imposed by these two layers of ABS and how we tackled them using the features of Java 8 and significantly changed the old Java backend results presented in Appendix A.4.

2.1.1 ABS Functional Layer

The functional programming paradigm that is available in ABS is very difficult to translate into executable code in Java. First we have the issue of data types which in a functional language can be defined as predicates whose recursion can extend infinitely. Evaluating such expressions in Java is not possible as Java does not have a lazy-evaluation mechanism and this severely affects performance and even breaks the program. Another issue is the pattern matching instructions which, even assuming it passes the evaluation issue mentioned before, still requires every data type declaration to translate into a Java class and every data definition into a Java object resulting in a very inefficient program memory-wise. This limits the support of ABS functional layer. We have to keep in mind that Java is a widely used programming language by software developers. While we want to use ABS to significantly reduce bugs and easy design through formal verification, we want to have generated code that is comparable and usable in terms of performance and resource usage to code written directly in Java. However, limiting the features translated from ABS means the

code can no longer be validated by static analysis tools and formally verified. This particular problem enforces even further the development of a Foreign Language Interface such that ABS can directly use constructs from its backends that bypass the type checking and formal verification tests. Therefore ABS features can be fully supported and benefit from formal verification tools and certain lower level implementations of the program can be ignored and left to each backend's compiler to detect any errors.

2.1.2 Cooperative Scheduling

In the ABS language, the core language contains constructs for the two finest levels of granularity in parallel computing which are scheduling method calls within an object and scheduling objects within a task. To model this cooperative scheduling as used by ABS in Java we need to use Threads to map each method call and a Thread pool to map each object. Therefore a construct of an asynchronous method call requires the creation of a new Thread inside a thread pool along with the start of this thread executing the method. Our research question results from the fact that each object is supposed to execute on one thread. In Java each object will be in fact a Thread pool containing the number of threads corresponding to the number of method calls invoked by the object to support that finest level of scheduling featured by ABS. This is done despite only one method running at most on that object. This underlying thread model significantly affects performance of an ABS-modelled application that is translated into Java code.

Java 8 new features allow wrapping of method calls into lightweight lambda expressions such that they can be put into a scheduling queue of a ForkJoinPool to which the running objects are mapped, significantly reducing the number of idle Threads at runtime. We chose this data structure over the more general Executor Services due its suitability to numerous small tasks that they have to process. The only drawback is that if a method call contains a recursive stack of synchronous calls, upon preemption this stack needs to be saved, a scenario which cannot be achieved through lambda expressions. To solve this issue we divided our research in two directions:

- On every preemption, we calculate the continuation and enqueue the rest of the call into the message queue.
- On every preemption, we try to optimally suspend the message thread until the continuation of the call is released.

The first direction is still in progress, as we have a fundamental technical limitation in the JVM/compiler. It is difficult to ask the JVM to make the rest of a method turn into another method call in the object's message queue. The second direction however offers the following possible scenario:

- Each asynchronous call/invocation is a message delivered in the corresponding object's queue.
- All objects in the same Concurrent Object Group (COG) compete for one Thread.
- A Sweeper Thread decides which task should be created and be available for execution from the available queues.
- A thread pool executes available tasks based on a work stealing mechanism.

The only challenge now is when an execution of multiple synchronous calls is preempted. This results in a call stack and context that need to be saved within a thread. To do this the executing thread from the pool is suspended and will compete again, upon release, with the other available tasks in the COG for selection by the Sweeper Thread to be made available to the thread pool.

2.2 Architectural Overview

In this section we present a high-level description of the Java backend used to generate executable code from ABS to Java. The motivation behind using Java is because it is one of the mainstream object oriented programming languages in terms of usability and ease of learning. Its usage covers several domains

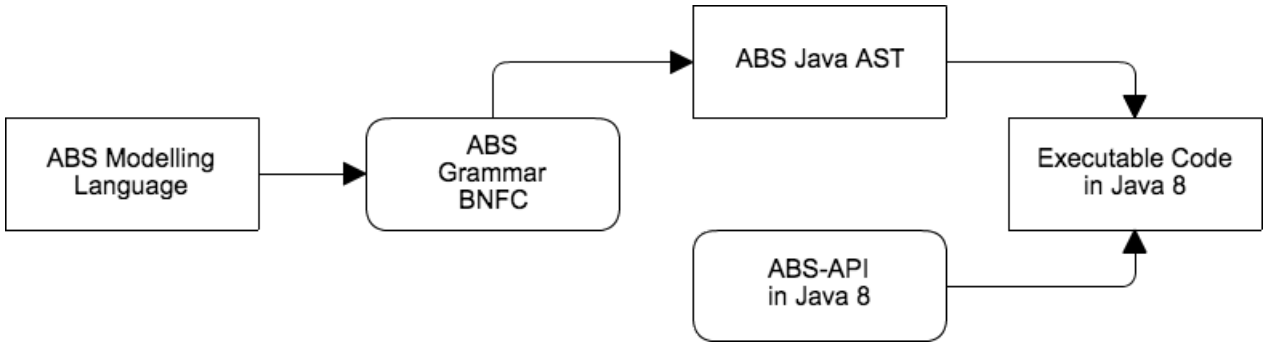

Figure 2.1: Code Generation Flow

within Computer Science such as mobile development, embedded systems, high performance applications, distributed web services and cloud computing. Java has support for both a concurrent model and remote method invocation standard to facilitate the translation of ABS's object-oriented layer and concurrency model. Oracle's Java 8 has also introduced modeling functions as lambda expressions, making it easier and more intuitive to map ABS's functional layer into Java. Finally, ABS further presents features for cloud resource management in terms of deployment components for which Java has libraries to support communication with the most important cloud APIs.

In order to have a stable and reliable code generation tool using the Java backend, it is clear that we require an aggregation of a lot of built-in, as well as external libraries that already exist in Java 8. To this end, we developed an extra module into the code generation tool, named the ABS-API which provides a wrapper over a lot of the libraries that we need. This API provides a set of operations that have a one-to-one translation to ABS features. Figure 2.2 presents an overview of the control flow of the code generation tool. With this architecture that aggregates Java 8 features to support ABS layers of programming, an interesting research question would be to use reverse engineering techniques for automated model extraction for a large-scale application written directly in Java code.

The control flow uses two main modules:

- The BNFC tool : that uses the ABS grammar to generate the AST in the Java language.
- The ABS-API library: that is used by the compiler to generate constructs in the final source files.

The translation process goes though 3 steps:

1. Parsing the ABS source file written using ABS language semantics and generation of the ABS AST.
2. The traversal of the AST that was generated in the Java language.
3. The construction of the Java source file with the aid of the ABS-API library in Java 8.

2.3 Benchmarks

Using this improved architecture for generating code in Java 8, we benchmarked our tool with the tests that are currently supported. The results are presented in Figure 2.2 in comparison with the performance times obtained with those in Appendix A.4 for the old Java backend. We observe a significant improvement in all tests performed, with 5 out of 16 tests significantly completing faster due to the lack of CPU or memory overhead. This initial result validates our approach and supports the use of Java to translate ABS code. With this new solution we can generate efficient production code that does not affect the program's performance at runtime despite the limitations of Java for functional programming or its expensive thread model.

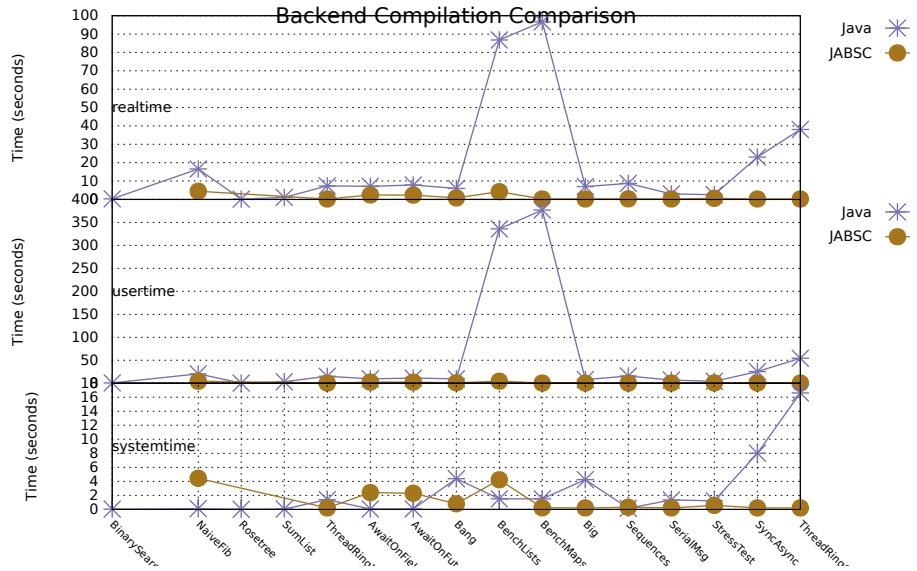

Figure 2.2: Java Backend (jabsc) Benchmarks

Chapter 3

Haskell Backend

In this section, we introduce another backend approach for ABS that predates the Java backend introduced before in Chapter 2. This ABS backend targets the Haskell programming language: Haskell is a purely-functional language with a by-default lazy evaluation strategy that employs static typing with both parametric and ad-hoc polymorphism. Haskell is widely known in academia and the language makes everyday more and more appearances in industry too [1], attributed to the fact that Haskell offers a good compromise between execution speed and abstraction level. An example of a successful tool built exclusively in Haskell is the BNF Converter (BNFC) which generates lexers and parsers for multiple languages (Java, Haskell, C++, ...) solely from a BNF grammar. We ourselves make use of the BNFC compiler tool for our ABS-to-Haskell backend, which was later adopted also by the ABS-to-Java backend as can be seen in Figure 1.1.

Motivation and Challenges When starting off the ABS-to-Haskell backend, the initial motivation was to develop a backend that can generate more efficient executable code compared to the reportedly slower at the time Maude and ex-Java ABS backends, which, in retrospect, are more appropriate for simulating and debugging ABS code than running it in production.

The translation of ABS to Haskell was relatively straightforward since the languages share many similarities, with the exception being the OO and subtype polymorphism that remained a particular challenge (see Section 3.1). After completing the implementation of the full ABS standard (which was the result of the previous HATS EU project) we extended the language with exceptions and preliminary support for Deployment Components in the Cloud (a goal of the current Envisage EU project). For this Cloud extension we were motivated by the fact, Haskell's programming model adheres to data immutability and "share-nothing" ideologies, which potentially deems Haskell as a better fit for transitioning ABS to the "Cloud".

Advantages The improved execution speed and the extension to the Cloud are the two main advantages that come with the ABS-to-Haskell backend, as it can be witnessed by the benchmarks in the Appendix A.4 and experimental results in Appendix A.2 respectively. The concurrency/threading model of Haskell proved to be well-suited for ABS' cooperative multitasking. On another level, by using Haskell our compiler infrastructure does not require anymore implementing a separate ABS typechecker and, instead the underlying Haskell typechecker can be used in-place for type-checking ABS programs. We further exploit the Haskell type-checker for (partially) inferring types in ABS programs, since Haskell has built-in support for Hindley-Milner type inference. Haskell is *purely* functional, and as such, the ABS-translated code is guaranteed (by the type-system) to not mix side-effectful object-oriented code inside purely-functional ABS code. Finally, since both languages are very similar and stay on the same (high) level of abstraction, it enabled us to prove the correctness and resource preservation of this ABS-to-Haskell translation, and is detailed in Chapter 4.

[1] `https://wiki.haskell.org/Haskell_in_industry`

3.1 Translating to Haskell

Our compiler, which is written in Haskell itself, translates input ABS programs to Haskell equivalent programs. This translation is very straightforward for a number of reasons: 1) the two languages are more or less the same at their core, i.e. purely-functional languages with ADTs and parametric-polymorphism; 2) ABS interfaces are simply a subset of Haskell's typeclasses (ad-hoc polymorphism); 3) ABS classes become algebraic datatypes (ADTs) acting as records (containers) of their fields, and objects become merely values of such types; 4) the bodies of methods (statements) become side-effectful (monadic) code. As mentioned earlier, mixing monadic code in pure code is disallowed by Haskell; 5) ABS modules simply become Haskell modules as the original ABS module system was inspired by Haskell's. We extend standard ABS with equivalent Haskell features, i.e. type inference, parametric type synonyms, exceptions-as-datatypes and we modify the past Foreign Function Interface (specifically designed Java with new syntactic and semantic support for interfacing to Haskell libraries: the user has to simply prefix an import declaration with `foreign`, e.g. `foreign import IOArray from Data.Array.IO`. This "foreign-importing" can be achieved because ABS and Haskell have the same calling conventions and 1-to-1 datatatype correspondence.

Despite the straightforward translation, a major challenge was to implement ABS' nominal subtyping: an ABS object with (interface) type $I2$ can be upcasted to any of its parent interfaces $I1$ at runtime, thus restricting its API (interface) exposal, as in the example:

```
{ I1 o1 = new C1();
  I2 o1_ = new C1();
  I2 o2 = new C2();
  List<I1> s = list[o1, o1_, o2]; }
```

In the above block, list `s` contains three objects, the first two instantiated by the same class, but typed with a different interface, whereas the second two are of the same interface but different class. However, since Haskell lists are homogeneous and Haskell is lacking builtin support for any subtyping, we have to do a type "trickery" to allow this list expression to type check. A `new` expression returns an *object reference* (ObjectRef) to the class' record. Thus the new expressions at lines 1 and 2 are typed in Haskell by (ObjectRef C1) and (ObjectRef C2) respectively. We create a *wrapper datatype* that instead types them by interface, hiding inside their class type.

```
data I2 = ∀ a . (I2_ a) ⇒ I2 (ObjectRef a)
```

In Haskell, this is called (despite the ∀ construct) an existential type. Now `o1` and `o2` are both typed by `I2`. The third object however is typed by the supertype interface `I1`. For that reason we have to upcast all subtypes; we do that with the method *up* defined in the typeclass `Sub` as. We not forget to upcast also the equality function (`==`), which checks for object reference equality, and not structural equality.

```
class Sub sub sup where
    up :: sub → sup
instance Sub I1 I1 where
        up x = x
instance Sub I2 I2 where
        up x = x
instance Sub I2 I1 where
        up (I2 a) = I1 a
```

This approach to nominal subtyping for Haskell leads to the generation of few boilerplate code and insignificant runtime performance overhead, as the places that subtyping can occur in ABS is limited (mainly, assignment and argument passing).

The generated Haskell code is subsequently typechecked by a standard Haskell compiler without the need of an external ABS typechecker; thus the final program will always be ABS-type safe, in the sense that all type errors are caught at compile time and no type-error escapes to runtime. However, a specialized ABS typechecker (as the one provided in the original abstools suite: `https://github.com/abstools/abstools`) may yield more precise and user-friendly type-error messages than the "general" Haskell typechecker; in other words, the Haskell typechecker cannot be fully aware of all the ABS language constructs. The ABS-to-Haskell

compiler is actively developed and situated at `https://github.com/bezirg/abs2haskell` with including installation instructions.

3.2 Parallel Runtime

The translated Haskell code is linked against our custom concurrent runtime framework, based on GHC's (Glasgow Haskell Compiler) runtime system (RTS), featuring SMP-enabled threads (Symmetric MultiProcessing parallelism, also known as multicore). Each alive Concurrent Object Group (COG) becomes simply a separate Haskell thread; Haskell's threads are very lightweight (green threads, i.e. small memory footprint), thus enabling us to spawn millions of COGs inside a single machine.

Each COG-thread retains an ABS process queue that holds processes to be executed; a new ABS process is created and put in the end of the queue upon each asynchronous method call. Processes are implemented as coroutines (which are themselves implemented as first-class continuations) and not as threads, something that allows us to store them inside the process queue as data. A continuation is a data-structure that contains the current execution state of the program (program counter, local variables, and the call stack) that when invoked, will replace the current state of the program with the continuation's saved state. This allows us to fully implement the cooperative multitasking of the ABS language: when a running process calls `suspend` or `await` its continuation will be captured and stored in the end of the queue; the COG will then pick the next (suspended) process from the head of the queue, essentially cooperatively (willfully) letting another process to execute.

This concurrency scheme based on continuations suffices to model ABS cooperative multitasking; however, it will result in needless resumption of processes that are not enabled; for example, consider a process that is suspended by calling `await f?` on a future f and put at the end of the COG's process queue. After a while, it will end up in the head of the queue where the COG will resume its execution, only to find out that the process has to be suspended again, since the future has not been resolved yet. This situation is more known by the term *busy-wait polling* and applies not only for futures but also when awaiting for arbitrary boolean expressions too. To alleviate busy-wait polling, we use a so-called process-disabled table where processes (continuations) are stored if they are known that their resumption cannot continue. The COG will not reschedule a disabled process of this table until its dependencies (future or boolean expression) are resolved. A (possibly different) COG will later inform this COG that some dependency (again, future or boolean expression) has been resolved and instruct the COG to update its process-disabled table based on the new information. The COG will then re-enable processes that have resolved all their dependencies and put them back in the end of its process queue for later resumption. This implementation technique avoids busy-wait polling, thus improving execution speed.

3.3 Cloud Runtime

We extend the parallel ABS-to-Haskell runtime with support for Deployment Components that provide a suitable abstraction of the Virtual Machines provided by the Cloud and which allow the application to distribute itself among multiple machines. The ABS programmer can dynamically create, monitor and shutdown such Deployment Components (Virtual Machines) and most importantly assign new objects to them. As such, an ABS cloud-application is consisted by a bunch of inter-VM communicating objects, effectively forming a distributed-object system which can control its own deployment and still benefit from the (local) parallelism.

The distributed communication of ABS processes is realized by Cloud-Haskell, which is a Haskell library for type-safe, fault-tolerant distributed programming. The distribution model of Cloud-Haskell resembles that of the Erlang programming language with the difference being that Cloud-Haskell has extra support for type-safe and version-safe message passing, a feature that we also make very much use in our Cloud Runtime extension. The resulted remote communication remains transparent to the user: new objects can be remotely created inside a different machine and asynchronous calls be made to remote objects (living inside

a remote machine) without changing the syntax and semantics of the ABS language. To achieve this, all ABS data (both datatypes and objects, passed to arguments) as well as methods have to be serialized before transmitting and deserialized at the other end. Primitive values and ADTs can be automatically serialized by Haskell; object and future references are serialized to proxy references which contain an ID uniquely identified across the distributed system (network). while leaving out their actual attributes/future results. Thanks to Cloud-Haskell, an asynchronous method call can be easily serialized to a so-called static closure, a datastructure that contains a known-at-compile-time code-pointer to the method and the serialization of all of its arguments. This implementation approach does not transfer the method-body's code (source-, byte- or machine-code).

The ABS user can create new Deployment Components (machines) just as creating objects (since DCs are modelled as objects). The DC class that is chosen dictates what kind of machine will be created; we currently provide library support 3 DC classes talking to 3 different providers: OpenNebula, Microsoft Azure and Local (similar to Docker containers). The network communication is left to Cloud-Haskell and is provider-dependent: OpenNebula and Azure with TCP and Local with in-memory transport. We plan to extend our library with support for more (cloud) providers.

Currently we are investigating the migration of ABS processes between DCs (machines); this can theoretically be achieved since ABS processes are merely data, and thus can be serialized and remotely transferred (migrated) from machine to machine. Our support for Deployment Components follows the syntax and semantics as it was first defined in the Deliverable D1.1 "Modeling of Systems"; the Appendix A.3 contains more information about the current implementation. We shortly plan to adopt the revised and more elaborate approach to Deployment Components (dynamic deployment) as defined in the recent Deliverable D1.3.1 "Modeling of Deployment".

3.4 Benchmarks

We compared all the current ABS backends for their execution speed and memory consumption in a series of sequential and parallel ABS programs that try to cover all aspects of the ABS language. These synthesized ABS benchmarks programs can be found at `https://github.com/abstools/abs-bench`). The benchmark results that are laid out in Appendix A.4 indicate that the Haskell backend is the fastest both in terms of elapsed time and memory residency. Specifically, the Haskell backend is on average **13x** faster while taking up **15x** less memory than the new Java backend; this may be attributed to the fact that the new Java backend relies on Java's heavyweight threads. Two other downsides of the new Java backend is that, firstly, it currently does not support (user-defined) algebraic datatypes (hence the `err` in the results table) and, secondly, it suffers from process starvation: there are certain correct ABS programs that terminate but unfortunately in the new Java backend they hang, because the employed threading model (static threadpool) limits how many "processor" units (COGs) can run concurrently. The old Java backend is slower than the new Java backend, and consequently slower than the Haskell backend (**256x** more time and **84x** more memory); the reason may be attributed to factors affecting also the new Java backend and also the fact that the old Java backend uses spin-waiting when monitoring active objects for their *await* conditions. The Erlang backend takes **596x** more time and **17x** more memory than the Haskell backend, since the backend follows the apparently slower, process-oriented approach, i.e. each ABS process is implemented as a separate lightweight thread: the COG's ABS processes are sitting in a token ring—the process holding the token can execute unless it is blocked in which case the token is passed that may cause needless spinning in certain cases. The Maude backend is extremely slow compared to all other backends since it is an interpreter, but surprisingly consumes comparable memory to Haskell (**9x** more memory than Haskell), and even in some cases less memory than the other 3 backends: new-,old- Java and Erlang.

Chapter 4

Resource Preservation

In this section we focus on a strong feature of our translation: the Haskell-translated program preserves the *resource consumption* of the original ABS program. In order to formalize this result we consider only the concurrent-object subset of the ABS language, since the functional part of ABS has a one-to-one straightforward correspondence to Haskell. Concretely, we consider the following simplified syntax:

$$\begin{array}{lll} S & ::= & x\texttt{:=}E \mid f\texttt{:=}x\texttt{!}m(\bar{z}) \mid \texttt{await}\ f \mid \texttt{skip} \mid \texttt{return}\ z \mid S_1;S_2 \\ E & ::= & x \mid r \mid \texttt{new} \mid f\texttt{.get} \mid m(\bar{z}) \\ D & ::= & m(\bar{y}) \mapsto S \\ P & ::= & \overline{D} : S \end{array}$$

Variables are fields (x) or futures (f), and the values are objects. A program is a sequence of method declarations followed by a main block. A statement is an assignment, an asynchronous call on an object ($x\texttt{!}m(\bar{z})$), an `await` on a future, `skip`, `return`ing a value or the composition of two statements. Assignments can store the value of a field (x), a constant reference (r), a new object (`new`) or the result of synchronous calls ($m(\bar{z})$) or asynchronous calls linked to a future ($f\texttt{.get}$). Both synchronous and asynchronous calls accept parameters ($\bar{z}$) that are considered as constant references.

The results are stated based on a simplified compilation from ABS to Haskell that can be found in `https://github.com/bezirg/abs2haskell-pure/tree/ast`. This compilation follows the same ideas as the full-blown Haskell backend presented in Chapter 3 but focused on the reduced language—see details in the attached article [4]. The compilation translates each ABS method into a Haskell method, and run-time configurations are represented as pairs (h, l) where h is a heap containing all the information of objects (both variables and process queues) and l is a list of active object identifiers. The main piece of this compilation is the `eval` Haskell function, that given a run-time configuration (h, l) and an object identifier o performs one step of evaluation. Function `eval` computes (h', l') where h' is the modified heap and l' is the new list of active objects. Using the `eval` function we can start from an initial run-time configuration (h_1, l_1) and chain $n-1$ steps of evaluation, creating an evaluation trace $\mathcal{T}_E \equiv (h_1, l_1) \rightharpoonup^{o_1}_{s_1} (h_2, l_2) \rightharpoonup^{o_2}_{s_2} \ldots \rightharpoonup^{o_{n-1}}_{s_{n-1}} (h_n, l_n)$. The decorations in the $\rightharpoonup$ arrow indicate the object o_i and the statement s_i executed by `eval`.

The first step to relate the ABS semantics and the `eval` function is to define a translation from Haskell run-time configurations (h, l) to ABS configurations A. We will consider the ABS semantics presented in [1], written $A_1 \rightsquigarrow A_2$, and the translation $\langle\!\langle (h, l) \rangle\!\rangle = A$ defined in the attached paper [4]. It is also important the notion of *cost model* (reported in Deliverable D3.3.2) to parameterize the type of resource we want to bound. Cost models are functions from ABS statements to real numbers, i.e., $\mathcal{M} : S \to \mathbb{R}$ that define different resource consumption measures. For instance, if the resource to measure is the number of executed steps, $\mathcal{M} : S \to 1$ such that each instruction has cost one. However, if one wants to measure memory consumption, we have that $\mathcal{M}(new) = \texttt{obj\_size}$, where `obj_size` refers to the size of an object reference, and $\mathcal{M}(instr) = 0$ for all remaining instructions. The resource preservation is based on the following notion of *cost*: given a concrete cost model $\mathcal{M}$, an object reference o and a program execution $\mathcal{T} \equiv A_1 \rightsquigarrow^{o_1}_{S_1} A_2 \rightsquigarrow^{o_2}_{S_2} \ldots \rightsquigarrow^{o_{n-1}}_{S_{n-1}} A_n$, the cost of the trace $\mathcal{C}(\mathcal{T}, o, \mathcal{M})$ is defined as follows (see also Deliverable

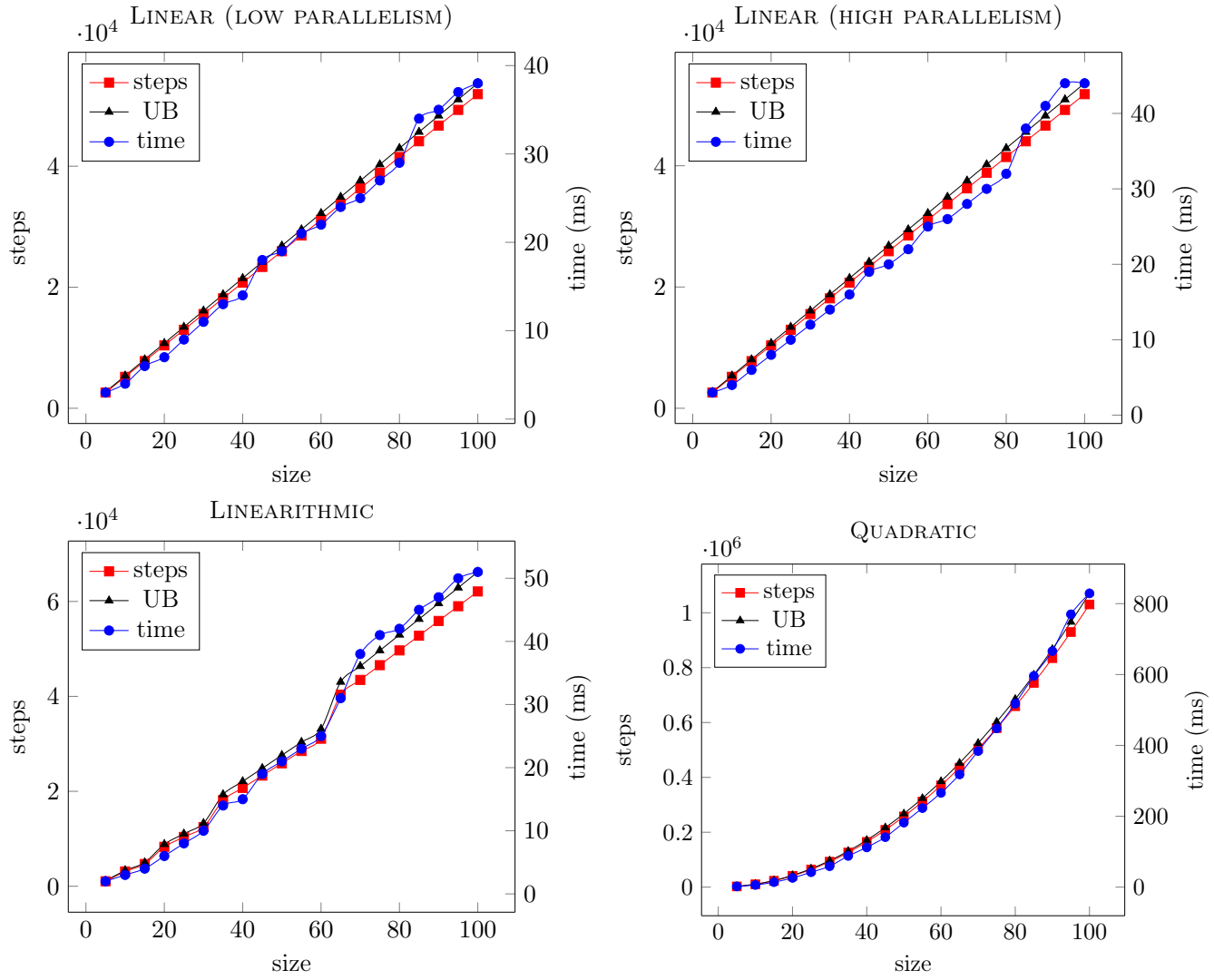

Figure 4.1: Execution steps vs. time (Intel® Core™ i7-4790 at 3.60GHz, 16 GB).

D3.3.2)

$$\mathcal{C}(\mathcal{T}, o, \mathcal{M}) = \sum_{t \in \mathcal{T}|_{\{o\}}} \mathcal{M}(t)$$

Notice that, from all the steps in the trace $\mathcal{T}$, it takes into account only those performed in object o, so the cost notion is *object-sensitive*.

In order to prove the preservation of the resource consumption in the compiled program, we prove that for any sequence of `eval` steps $\mathcal{T}_E$ there is also an equivalent execution trace $\mathcal{T}$ in the ABS semantics with the same cost, i.e., $\mathcal{C}(\mathcal{T}_E, o, \mathcal{M}) = \mathcal{C}(\mathcal{T}, o, \mathcal{M})$.

Theorem 4.0.1 (Consumption Preservation) *Let* $\mathcal{T}_E \equiv (h_1, l_1) \rightharpoonup_{s_1}^{o_1} (h_1, l_2) \rightharpoonup_{l_2}^{o_2} \ldots \rightharpoonup_{s_{n-1}}^{o_{n-1}} (h_n, l_n)$ *and evaluation trace. Then there is a trace* $\mathcal{T} = \langle\!\langle (h_1, l_1) \rangle\!\rangle \rightsquigarrow^* \langle\!\langle (h_n, l_n) \rangle\!\rangle$ *such that* $\mathcal{C}(\mathcal{T}_E, o, \mathcal{M}) = \mathcal{C}(\mathcal{T}, o, \mathcal{M})$.

As a side effect of the previous theorem, we know that the upper bounds that are inferred from the ABS programs using the resource analyzer SACO are valid upper bounds for the Haskell translated code. Let $UB_{main}()|_o$ be the upper bound obtained for a program starting from the main block, restricted to the object o–see details in [1]. Then we can state the following resource preservation result:

Corollary 4.0.2 (Bound preservation) *Let $\mathcal{T}_E$ be an evaluation trace starting from the main block. Then* $\mathcal{C}(\mathcal{T}_E, o, \mathcal{M}) \leq UB_{main}()|_o$

Besides proving that the execution of compiled Haskell programs has the same resource consumption as the original ABS traces (i.e., they execute the same statements in the same order and in the same objects), it is important that the compilation does not introduce an overhead during execution so that run-times are proportional to the steps executed. In order to evaluate this hypothesis, we have created some programs (see [4]) with different asymptotic costs and measured the number of statements executed (steps) and their run-time. These benchmark programs create a number n of objects (size) and invoke some tasks in each one: one task for the linear programs, *log* n tasks for the linearithmic program and n tasks for the quadratic program. The difference between the two linear programs is that the *low parallelism* version awaits for the result of the task before creating the next object, whereas the *high parallelism* version does not await. Fig. 4.1 shows the results of the tests as graphs, where the left vertical axis is used for the number of steps and the right vertical axis for the run-time in milliseconds. The graphs show that both the steps and time plots have the same growth rate in all the programs thus confirming the proportionality, i.e., the execution of one statement requires a constant amount of time. We have added to the graphs the resource bounds (UB) obtained by the SACO tool (see Deliverable D3.3.2) in the original ABS programs using the cost model that measures the number of statements executed. As can be appreciated, the bounds are higher than the actual number of steps but they are very precise for all the programs, with only a small difference in the number of steps. This small imprecision in the upper bounds is caused by the constructors methods: the subset of the ABS language presented in this chapter does not include constructors, but full ABS (and the SACO tool as well) considers that every object has a constructor. Therefore, the SACO tool will count a constant number of extra steps whenever a new object is created, corresponding to the invocation and execution of the implicit constructor. Thus, our experiments show that the upper bounds obtained by SACO are accurate approximations of the actual steps performed and have a direct correlation to the actual execution time using the Haskell backend.

Bibliography

[1] Elvira Albert, Puri Arenas, Jesús Correas, Samir Genaim, Miguel Gómez-Zamalloa, Germán Puebla, and Guillermo Román-Díez. Object-Sensitive Cost Analysis for Concurrent Objects. *Software Testing, Verification and Reliability*, 25(3):218–271, 2015.

[2] Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, Germán Puebla, and Guillermo Román-Díez. SACO: Static Analyzer for Concurrent Objects. In *20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 8413 of *Lecture Notes in Computer Science*, pages 562–567. Springer-Verlag, 2014.

[3] Elvira Albert, Puri Arenas, Miguel Gómez-Zamalloa, and Jose Miguel Rojas. Test Case Generation by Symbolic Execution: Basic Concepts, a CLP-Based Instance, and Actor-Based Concurrency. In Marco Bernardo, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen, and Ina Schaefer, editors, *Formal Methods for Executable Software Models*, volume 8483 of *Lecture Notes in Computer Science*, pages 263–309. Springer-Verlag, June 2014.

[4] Elvira Albert, Nikolaos Bezirgiannis, Frank de Boer, and Enrique Martin-Martin. A Formal, Resource Consumption-Preserving Translation of Actors to Haskell. Submitted to FLOPS 2016.

[5] Elvira Albert, Jesús Correas, Germán Puebla, and Guillermo Román-Díez. A Multi-Domain Incremental Analysis Engine and its Application to Incremental Resource Analysis. *Theoretical Computer Science*, 585:91–114, 2015.

[6] Nikolaos Bezirgiannis and Frank S. de Boer. ABS: a high-level modeling language for Cloud-Aware Programming. In *Proc. SOFSEM '16*. Springer, 2016. To appear.

[7] Crystal Chang Din, Richard Bubel, and Reiner Hähnle. KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In Amy P. Felty and Aart Middeldorp, editors, *Proceedings of the 25th International conference on Automated Deduction*, volume 9195 of *Lecture Notes in Computer Science*, pages 517–526. Springer-Verlag, 2015.

[8] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer-Verlag, 2011.

[9] Behrooz Nobakht and Frank S. de Boer. Programming with actors in Java 8. In Tiziana Margaria and Bernhard Steffen, editors, *6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA'14)*, volume 8803 of *Lecture Notes in Computer Science*. Springer-Verlag, 2014.

Glossary

ForkJoinPool A Thread pool that employs work-stealing:all threads in the pool attempt to find and execute tasks submitted to the pool and/or created by other active tasks (eventually blocking waiting for work if none exist)

Concurrent Object Group An abstraction in ABS that groups together several active objects that are all scheduled to execute on a single thread.

Appendix A

Papers

A.1 Programming with Actors in Java 8

Programming with actors in Java 8*

Behrooz Nobakht[1,2] and Frank S. de Boer[3]

[1] Leiden Advanced Institute of Computer Science
Leiden University
bnobakht@liacs.nl
[2] SDL Fredhopper
bnobakht@sdl.com
[3] Centrum Wiskunde en Informatica
frb@cwi.nl

Abstract. There exist numerous languages and frameworks that support an implementation of a variety of actor-based programming models in Java using concurrency utilities and threads. Java 8 is released with fundamental new features: lambda expressions and further dynamic invocation support. We show in this paper that such features in Java 8 allow for a high-level actor-based methodology for programming distributed systems which supports the programming to interfaces discipline. The embedding of our actor-based Java API is shallow in the sense that it abstracts from the actual thread-based deployment models. We further discuss different concurrent execution and thread-based deployment models and an extension of the API for its actual parallel and distributed implementation. We present briefly the results of a set of experiments which provide evidence of the potential impact of lambda expressions in Java 8 regarding the adoption of the actor concurrency model in large-scale distributed applications.

Keywords: Actor model, Concurrency, Asynchronous Message, Java, Lambda Expression

1 Introduction

Java is beyond doubt one of the mainstream object oriented programming languages that supports a *programming to interfaces* discipline [9,35]. Through the years, Java has evolved from a mere programming language to a huge platform to drive and envision standards for mission-critical business applications. Moreover, the Java language itself has evolved in these years to support its community with new language features and standards. One of the noticeable domains of focus in the past decade has been distribution and concurrency in research and application. This has led to valuable research results and numerous libraries and frameworks with an attempt to provide distribution and

* This paper is funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services, http://www.envisage-project.eu.

concurrency at the level of Java language. However, it is widely recognized that the thread-based model of concurrency in Java that is a well-known approach is not appropriate for realizing distributed systems because of its inherent synchronous communication model. On the other hand, the event-driven actor model of concurrency introduced by Hewitt [17] is a powerful concept for modeling distributed and concurrent systems [2,1]. Different extensions of actors are proposed in several domains and are claimed to be the most suitable model of computation for many applications [18]. Examples of these domains include designing embedded systems [25,24], wireless sensor networks [6], multi-core programming [22] and delivering cloud services through SaaS or PaaS [5]. This model of concurrent computation forms the basis of the programming languages Erlang [3] and Scala [16] that have recently gained in popularity, in part due to their support for scalable concurrency. Moreover, based on the Java language itself, there are numerous libraries that provide an implementation of an actor-based programming model.

The main problem addressed in this paper is that in general existing actor-based programming techniques are based on an explicit encoding of mechanisms at the application level for message passing and handling, and as such overwrite the general object-oriented approach of method look-ups that forms the basis of programming to interfaces and the design-by-contract discipline [26]. The entanglement of event-driven (or asynchronous messaging) and object-oriented method look-up makes actor-based programs developed using such techniques extremely difficult to reason about and formalize. This clearly hampers the promotion of actor-based programming in mainstream industry that heavily practices object-oriented software engineering.

The main result of this paper is a Java 8 API for programming distributed systems using asynchronous message passing and a corresponding actor programming methodology which abstracts invocation from execution (e.g. thread-based deployment) and fully supports programming to interfaces discipline. We discuss the API architecture, its properties, and different concurrent execution models for the actual implementation.

Our main approach consists of the explicit description of an actor in terms of its *interface*, the use of the recently introduced lambda expressions in Java 8 in the implementation of asynchronous message passing, and the formalization of a corresponding high-level actor programming methodology in terms of an executable modeling language which lends itself to formal analysis, ABS [20].

The paper continues as follows: in Section 2, we briefly discuss a set of related works on actors and concurrent models especially on JVM platform. Section 3 presents an example that we use throughout the paper, we start to model the example using a library. Section 4 briefly introduces a concurrent modeling language and implements the example. Section 5 briefly discusses Java 8 features that this works uses for implementation. Section 6 presents how an actor model maps into programming in Java 8. Section 7 discusses in detail the implementation architecture of the actor API. Section 8 discusses how a number of benchmarks were performed for the implementation of the API and how

they compare with current related works. Section 9 concludes the paper and discusses the future work.

2 Related Work

There are numerous works of research and development in the domain of actor modeling and implementation in different languages. We discuss a subset of the related work in the level of modeling and implementation with more focus on Java and JVM-based efforts in this section.

Erlang [3] is a programming language used to build massively scalable soft real-time systems with requirements on high availability. Some of its uses are in telecoms, banking, e-commerce, computer telephony and instant messaging. Erlang's runtime system has built-in support for concurrency, distribution and fault tolerance. While threads require external library support in most languages, Erlang provides language-level features for creating and managing processes with the aim of simplifying concurrent programming. Though all concurrency is explicit in Erlang, processes communicate using message passing instead of shared variables, which removes the need for locks. Elixir [33] is a functional meta-programming aware language built on top of the Erlang VM. It is a dynamic language with flexible syntax with macros support that leverages Erlang's abilities to build concurrent, distributed, fault-tolerant applications with hot code upgrades.

Scala is a hybrid object-oriented and functional programming language inspired by Java. The most important concept introduced in [16] is that Scala actors unify *thread-based* and *event-based* programming model to fill the gap for concurrency programming. Through the event-based model, Scala also provides the notion of continuations. Scala provides quite the same features of scheduling of tasks as in concurrent Java; i.e., it does not provide a direct and customizable platform to manage and schedule priorities on messages sent to other actors. Akka [15] is a toolkit and runtime for building highly concurrent, distributed, and fault tolerant event-driven applications on the JVM based on actor model.

Kilim [31] is a message-passing framework for Java that provides ultra-lightweight threads and facilities for fast, safe, zero-copy messaging between these threads. It consists of a bytecode postprocessor (a "weaver"), a run time library with buffered mailboxes (multi-producer, single consumer queues) and a user-level scheduler and a type system that puts certain constraints on pointer aliasing within messages to ensure interference-freedom between threads. The SALSA [34,22] programming language (Simple Actor Language System and Architecture) is an active object-oriented programming language that uses concurrency primitives beyond asynchronous message passing, including token-passing, join, and first-class continuations.

RxJava [7] by Netflix is an implementation of reactive extensions [27] from Microsoft. Reactive extensions try to provide a solution for composing asynchronous and event-based software using observable pattern and scheduling.

An interesting direction of this library is that it uses reactive programming to avoid a phenomenon known as "callback hell"; a situation that is a natural consequence of composing `Future` abstractions in Java specifically when they wait for one another. However, RxJava advocates the use of asynchronous functions that are triggered in response to the other functions. In the same direction, LMAX Disruptor [4,8] is a highly concurrent event processing framework that takes the approach of event-driven programming towards provision of concurrency and asynchronous event handling. The system is built on the JVM platform and centers on a Business Logic Processor that can handle 6 million events per second on a single thread. The Business Logic Processor runs entirely in-memory using event sourcing. The Business Logic Processor is surrounded by Disruptors - a concurrency component that implements a network of queues that operate without needing locks.

3 State of the Art: An example

In the following, we illustrate the state of the art in actor programming by means of a simple example using the Akka [32] library which features asynchronous messaging and which is used to program actors in both Scala and Java. We want to model in Akka an "asynchronous ping-pong match" between two actors represented by the two interfaces `IPing` and `IPong` which are depicted in Listings 1 and 2. An asynchronous call by the actor implementing the `IPong` interface of the `ping` method of the actor implementing the `IPing` interface should generate an asynchronous call of the `pong` method of the callee, and vice versa. We intentionally design `ping` and `pong` methods to take arguments in order to demonstrate how method arguments may affect the use of an actor model in an object-oriented style.

Listing 1: Ping as an interface

```
1 public interface IPing {
2   void ping(String msg);
3 }
```

Listing 2: Pong as an interface

```
1 public interface IPong {
2   void pong(String msg);
3 }
```

To model an actor in Akka by a class, say `Ping`, with interface `IPing`, this class is required *both* to *extend* a given pre-defined class `UntypedActor` and *implement* the interface `IPing`, as depicted in Listings 3 and 4. The class `UntypedActor` provides two Akka framework methods `tell` and `onReceive` which are used to enqueue and dequeue asynchronous messages. An asynchronous call to, for example, the method `ping` then can be modeled by passing a user-defined encoding of this call, in this case by prefixing the string argument with the string "pinged", to a (synchronous) call of the `tell` method which results in enqueuing the message. In case this message is dequeued the implementation of the `onReceive` method as provided by the `Ping` class then calls the `ping` method.

Listing 3: Ping actor in Akka

```
1  public class Ping(ActorRef pong)
2    extends UntypedActor
3    implements IPing {
4
5    public void ping(String msg) {
6      pong.tell("ponged," + msg)
7    }
8
9    public void onReceive(Object m)
         {
10     if (!(m instanceof String)) {
11       // Message not understood.
12     } else
13     if (((String) m).startsWith("
           pinged") {
14       // Explicit cast needed.
15       ping((String) m);
16     }
17   }
18 }
```

Listing 4: Pong class in Akka

```
1  public class Pong
2    extends UntypedActor
3    implements IPong {
4
5    public void pong(String msg) {
6      sender().tell(
7        "pinged," + msg);
8    }
9
10   public void onReceive(Object m)
         {
11     if (!(m instanceof String)) {
12       // Message not understood.
13     } else
14     if (m.startsWith("ponged") {
15       // Explicit cast needed.
16       ping((String) m);
17     }
18   }
19 }
```

Access to the sender of the message in Akka is provided by `sender()`. In the main method as described in Listing 5 we show how the initialize and start the ping/pong match. Note that a reference to the "pong" actor is passed to the "ping" actor.

Listing 5: main in Akka

```
1 ActorSystem s = ActorSystem.create
      ();
2 ActorRef pong = s.actorOf(Props.
      create(Pong.class));
3 ActorRef ping = s.actorOf(Props.
      create(Ping.class, pong));
4 ping.tell(""); // To get a Future
```

Further, both the `onReceive` methods are invoked by Akka `ActorSystem` itself. In general, Akka actors are of type `ActorRef` which is an abstraction provided by Akka to allow actors send asynchronous messages to one another. An immediate consequence of the above use of inheritance is that the class `Ping` is now exposing a public behavior that is *not* specified by its *interface*. Furthermore, a "ping" object refers to a "pong" object by the type `ActorRef` . This means that the interface `IPong` is not directly visible to the "ping" actor. Additionally, the implementation details of receiving a message should be "hand coded" by the programmer into the special method `onReceive` to define the responses to the received messages. In our case, this implementation consists of a decoding of the message (using type-checking) in order to *look up* the method that subsequently should be invoked. This fundamentally interferes with the general object-oriented mechanism for method look-up which forms the basis of the programming to interfaces discipline. In the next section, we continue the same example and discuss an actor API for directly calling asynchronously methods using the general object-oriented mechanism for method look-up. Akka has recently released

a new version that supports Java 8 features [4]. However, the new features can be categorized as syntax sugar on how incoming messages are filtered through object/class matchers to find the proper type.

4 Actor Programming in Java

We first describe informally the actor programming model assumed in this paper. This model is based on the Abstract Behavioral Specification language (ABS) introduced in [20]. ABS uses asynchronous method calls, futures, interfaces for encapsulation, and cooperative scheduling of method invocations inside concurrent (active) objects. This feature combination results in a concurrent object-oriented model which is inherently compositional. More specifically, actors in ABS have an identity and behave as active objects with encapsulated data and methods which represent their state and behavior, respectively. Actors are the units of concurrency: conceptually an actor has a dedicated processor. Actors can only send asynchronous messages and have queues for receiving messages. An actor progresses by taking a message out of its queue and processing it by executing its corresponding method. A method is a piece of sequential code that may send messages. Asynchronous method calls use futures as dynamically generated references to return values. The execution of a method can be (temporarily) suspended by release statements which give rise to a form of cooperative scheduling of method invocations inside concurrent (active) objects. Release statements can be conditional (e.g., checking a future for the return value). Listings 7, 8 and 6 present an implementation of ping-pong example in ABS. By means of the statement on line 6 of Listing 7 a "ping" object directly calls asynchronously the `pong` method of its "pong" object, and vice versa. Such a call is stored in the message queue and the called method is executed when the message is dequeued. Note that variables in ABS are declared by interfaces. In ABS, `Unit` is similar to `void` in Java.

Listing 6: main in ABS

```
1 ABSIPong pong;
2 pong = new ABSPong;
3 ping = new ABSPing(pong);
4 ping ! ping("");
```

[4] Documentation available at `http://doc.akka.io/docs/akka/2.3.2/java/lambda-index-actors.html`

Listing 7: Ping in ABS

```
1 interface ABSIPing {
2   Unit ping(String msg);
3 }
4 class ABSPing(ABSIPong pong)
      implements ABSIPing {
5   Unit ping(String msg) {
6   pong ! pong("ponged," + msg);
7   }
8 }
```

Listing 8: Pong in ABS

```
1 interface ABSIPong {
2   Unit pong(String msg);
3 }
4 class ABSPong implements ABSIPong
      {
5   Unit pong(String msg) {
6     sender ! ping( "pinged," + msg
          );
7   }
8 }
```

5 Java 8 Features

In the next section, we describe how ABS actors are implemented in Java 8 as API. In this section we provide an overview of the features in Java 8 that facilitate an efficient, expressive, and precise implementation of an actor model in ABS.

Java Defender Methods Java *defender* methods (JSR 335 [13]) use the new keyword **`default`**. Defender methods are declared for **`interface`**s in Java. In contrast to the other methods of an interface, a default method is not an abstract method but must have an implementation. From the perspective of a client of the interface, defender methods are no different from ordinary interface methods. From the perspective of a hierarchy descendant, an implementing class can optionally *override* a default method and change the behavior. It is left as a decision to any class implementing the interface whether or not to override the default implementation. For instance, in Java 8 `java.util.Comparator` provides a default method `reversed()` that creates a reversed-order comparator of the original one. Such default method eliminates the need for any implementing class to provide such behavior by inheritance.

Java Functional Interfaces Functional interfaces and lambda expressions (JSR 335 [13]) are fundamental changes in Java 8. A **`@FunctionalInterface`** is an annotation that can be used for interfaces in Java. Conceptually, any class or interface is a functional interface if it consists of exactly one *abstract* method. A lambda expression in Java 8, is a runtime translation [11] of any type that is replaceable by a functional interface. Many of Java's classic interfaces are functional interfaces from the perspective of Java 8 and can be turned into lambda expressions; e.g. `java.lang.Runnable` or `java.util.Comparator`. For instance,

```
(s1, s2) → return s1.compareTo(s2);
```

is a lambda expression that can be statically cast to an instance of a `Comparator<String>`; because it can be replaced with a functional interface that has a method with two strings and returning one integer. Lambda expressions in Java 8 *do not* have an intrinsic type. Their type is bound to the context that they are used in but

their type is always a functional interface. For instance, the above definition of a lambda expression can be used as:

```
Comparator<String> cmp1 = (s1, s2) → return s1.compareTo(s2);
```

in one context while in the other:

```
Function<String> cmp2 = (s1, s2) → return s1.compareTo(s2);
```

given that `Function<T>` is defined as:

```
interface Function<T> { int apply(T t1, T t2); }
```

In the above examples, the same lambda expression is statically cast to a different matching functional interface based on the context. This is a fundamental new feature in Java 8 that facilitates application of functional programming paradigm in an object-oriented language.

This work of research extensively uses this feature of Java 8. Java 8 marks many of its own APIs as functional interfaces most important of which in this context are `java.lang.Runnable` and `java.util.concurrent.Callable`. This means that a lambda expression can replace an instance of `Runnable` or `Callable` at runtime by JVM. We will discuss later how we utilize this feature to allow us model an asynchronous message into an instance of a `Runnable` or `Callable` as a form of a lambda expression. A lambda expression equivalent of a `Runnable` or a `Callable` can be treated as a queued message of an actor and executed.

Java Dynamic Invocation Dynamic invocation and execution with method handles (JSR 292 [29]) enables JVM to support efficient and flexible execution of method invocations in the absence of static type information. JSR 292 introduces a new byte code instruction `invokedynamic` for JVM that is available as an API through `java.lang.invoke.MethodHandles`. This API allows translation of lambda expression in Java 8 at runtime to be executed by JVM. In Java 8, use of lambda expression are favored over anonymous inner classes mainly because of their performance issues [12]. The abstractions introduced in JSR 292 perform better that Java Reflection API using the new byte code instruction. Thus, lambda expressions are compiled and translated into method handle invocations rather reflective code or anonymous inner classes. This feature of Java 8 is indirectly use in ABS API through the extensive use of lambda expressions. Moreover, in terms of performance, it has been revealed that invoke dynamic is much better than using anonymous inner classes [12].

6 Modeling actors in Java 8

In this section, we discuss how we model ABS actors using Java 8 features. In this mapping, we demonstrate how new features of Java 8 are used.

The `Actor` Interface We introduce an interface to model actors using Java 8 features discussed in Section 5. Implementing an interface in Java means that the object exposes public APIs specified by the interface that is considered the behavior of the object. Interface implementation is opposed to inheritance extension in which the object is possibly forced to expose behavior that may not be part of its intended interface. Using an interface for an actor allows an object to preserve its own interfaces, and second, it allows for multiple interfaces to be implemented and composed.

A Java API for the implementation of ABS models should have the following main three features. First, an object should be able to send asynchronously an arbitrary message in terms of a method invocation to a receiver actor object. Second, sending a message can optionally generate a so-called future which is used to refer to the return value. Third, an object during the processing of a message should be able to access the "sender" of a message such that it can reply to the message by another message. All the above must co-exist with the fundamental requirement that for an object to act like an actor (in an object-oriented context) should *not* require a modification of its intended interface.

The `Actor` interface (Listings 9 and 10) provides a set of **`default`** methods, namely the `run` and `send` methods, which the implementing classes do not need to re-implement. This interface further encapsulates a queue of messages that supports concurrent features of Java API [5]. We distinguish two types of messages: messages that are not expected to generate any result and messages that are expected to generate a result captured by a future value; i.e. an instance of `Future` in Java 8. The first kind of messages are modeled as instances of `Runnable` and the second kind are modeled instances of `Callable`. The default `run` method then takes a message from the queue, checks its type and executes the message correspondingly. On the other hand, the default (overloaded) `send` method stores the sent message and creates a future which is returned to the caller, in case of an instance of `Callable`.

[5] Such API includes usage of different interfaces and classes in `java.util.concurrent` package [23]. The concurrent Java API supports blocking and synchronization features in a high-level that is abstracted from the user.

Listing 9: Actor interface (1)

```
1  public interface Actor {
2    public void run() {
3      Object m = queue.take();
4
5      if (m instanceof Runnable) {
6        ((Runnable) m).run();
7      } else
8
9      if (m instanceof Callable) {
10       ((Callable) m).call();
11     }
12   }
13
14   // continue to the right
```

Listing 10: Actor interface (2)

```
1
2    public void send(Runnable m) {
3      queue.offer(m);
4    }
5
6    public <T> Future<T>
7      send(Callable<T> m) {
8        Future<T> f =
9          new FutureTask(m);
10       queue.offer(f);
11       return f;
12   }
13 }
```

Modeling Asynchronous Messages We model an asynchronous call

$$\texttt{Future<V>}\ f = e_0\ !\ m(e_1, \ldots, e_n)$$

to a method in ABS by the Java 8 code snippet of Listing 11. The final local variables $u_1, \ldots, u_n$ (of the caller) are used to store the values of the Java 8 expressions $e_1, \ldots, e_n$ corresponding to the actual parameters $e_1, \ldots, e_n$. The types T_i, $i = 1, \ldots, n$, are the corresponding Java 8 types of e_i, $i = 1, \ldots, n$.

Listing 11: Async messages with futures

```
1 final T1 u1 = e1;
2 . . .
3 final Tn un = en;
4 Future<V> v = e0.send(
5   () → { return m(u1,...,un); }
6 );
```

Listing 12: Async messages w/o futures

```
1 final T1 u1 = e1;
2 . . .
3 final Tn un = en;
4 e0.send(
5   { () → m(u1,...,un); }
6 );
```

The lambda expression which encloses the above method invocation is an instance of the functional interface; e.g. `Callable`. Note that the generated object which represents the lambda expression will contain the local context of the caller of the method "m" (including the local variables storing the values of the expressions $e_1, \ldots, e_n$), which will be restored upon execution of the lambda expression. Listing 12 models an asynchronous call to a method without a return value.

As an example, Listings 13 and 14 present the running ping/pong example, using the above API. The main program to use ping and pong implementation is presented in Listing 15.

Listing 13: Ping as an Actor

```
1 public class Ping(IPong pong)
       implements IPing, Actor {
2  public void ping(String msg) {
3   pong.send( () → { pong.("ponged
        ," + msg) } );
4   }
5 }
```

Listing 14: Pong as an Actor

```
1 public class Pong implements IPong
      , Actor {
2   public void pong(String msg) {
3     sender().send(( ) → { ping.("
          pinged," + msg) } );
4   }
5 }
```

As demonstrated in the above examples, the "ping" and "pong" objects *preserve* their own *interfaces* contrary to the example depicted in Section 3 in which the objects *extend* a specific "universal actor abstraction" to inherit methods and behaviors to become an actor. Further, messages are processed *generically* by the `run` method described in Listing 9. Although, in the first place, sending an asynchronous may look like to be able to change the recipient actor's state, this is not correct. The variables that can be used in a lambda expression are *effectively* final. In other words, in the context of a lambda expression, the recipient actor only provides a snapshot view of its state that cannot be changed. This prevents abuse of lambda expressions to change the receiver's state.

Modeling Cooperative Scheduling The ABS statement **`await`** g, where g is a boolean guard, allows an active object to preempt the current method and schedule another one. We model cooperative scheduling by means of a call to the `await` method in Listing 16. Note that the preempted process is thus passed as an additional parameter and as such queued in case the guard is false, otherwise it is executed. Moreover, the generation of the continuation of the process is an optimization task for the code generation process to prevent code duplication.

Listing 15: main in ABS API

```
1 IPong pong = new Pong();
2 IPing ping = new Ping(pong);
3 ping.send(
4   () -> ping.ping("")
5 );
```

Listing 16: Java 8 await implementation

```
1 void await(final Boolean guard,
2            final Runnable cont) {
3   if (!guard) {
4     this.send(() →
5       { this.await(guard, cont) })
6   } else { cont.run() }
7 }
```

7 Implementation Architecture

Figure 1 presents the general layered architecture of the actor API in Java 8. It consists of three layers: the routing layer which forms the foundation for the support of distribution and location transparency [22] of actors, the queuing layer which allows for different implementations of the message queues, and

finally, the processing layer which implements the actual execution of the messages. Each layer allows for further customization by means of plugins. The implementation is available at `https://github.com/CrispOSS/abs-api`.

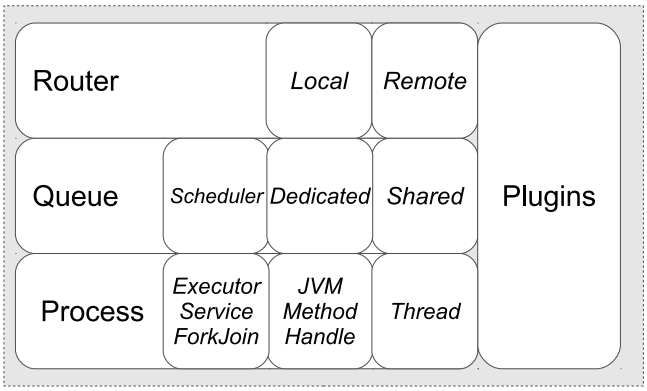

Fig. 1: Architecture of Actor API in Java 8

We discuss the architecture from bottom layer to top. The implementation of actor API preserves a faithful mapping of message processing in ABS modeling language. An actor is an active object in the sense that it controls how the next message is executed and may release any resources to allow for cooperative scheduling. Thus, the implementation is required to optimally utilize JVM threads. Clearly, allocating a dedicated thread to each message or actor is not scalable. Therefore, actors need to *share* threads for message execution and yet be in full control of resources when required. The implementation fundamentally separates *invocation* from *execution*. An asynchronous message is a reference to a method invocation until it starts its execution. This allows to minimize the allocation of threads to the messages and facilitates sharing threads for executing messages. Java concurrent API [23] provides different ways to deploy this separation of invocation from execution. We take advantage of Java Method Handles [29] to encapsulate invocations. Further we utilize different forms of `ExecutorService` and `ForkJoinPool` to deploy concurrent invocations of messages in different actors.

In the next layer, the actor API allows for different implementations of a queue for an actor. A dedicated queue for each actor simplifies the process of queuing messages for execution but consumes more resources. However, a shared queue for a set of actors allows for memory and storage optimization. This latter approach of deployment, first, provides a way to utilize the computing power of multi-core; for instance, it allows to use work-stealing to maximize the usage of thread pools. Second, it enables application-level scheduling of messages. The different implementations cater for a variety of

plugins, like one that releases computation as long as there is no item in the queue and becomes active as soon as an item is placed into the queue; e.g. `java.util.concurrent.BlockingQueue`. Further, different plugins can be injected to allow for scheduling of messages extended with deadlines and priorities [28].

We discuss next the distribution of actors in this architecture. In the architecture presented in Figure 1, each layer can be *distributed* independently of another layer in a transparent way. Not only the routing layer can provide distribution, the queue layer of the architecture may also be remote to take advantage of cluster storage for actor messages. A remote routing layer can provide access to actors transparently through standard naming or addresses. We exploit the main properties of actor model [1,2] to distribute actors based on our implementation. From a distributed perspective, the following are the main requirements for distributing actors:

Reference Location Transparency Actors communicate to one another using references. In an actor model, there is no in-memory object reference; however, every actor reference denotes a location by means of which the actor is accessible. The reference location may be local to the calling actor or remote. The reference location is *physically* transparent for the calling actor.

Communication Transparency A message m from actor A to actor B may possibly lead to transferring m over a network such that B can process the message. Thus, an actor model that supports distribution must provide a layer of remote communication among its actors that is transparent, i.e., when actor A sends message m, the message is transparently transferred over the network to reach actor B. For instance, actors existing in an HTTP container that transparently allows such communication. Further, the API implementation is required to provide a mechanism for serialization of messages. By default, every object in JVM cannot be assumed to be an instance of `java.io.Serializable`. However, the API may enforce that any remote actor should have the required actor classes in its JVM during runtime which allows the use of the JVM's general object serialization [6] to send messages to remote actors and receive their responses. Additionally, we model asynchronous messages with lambda expressions for which Java 8 supports serialization by specification [7].

Actor Provisioning During a life time of an actor, it may need to create new actors. Creating actors in a local memory setting is straightforward. However, the local setting *does* have a capacity of number of actors it can hold. When an actor creates a new one, the new actor may actually be initialized in a remote resource. When the resource is not available, it should be first provisioned. However, this resource provisioning should be transparent to the actor and only the eventual result (the newly created actor) is visible.

[6] Java Object Serialization Specification: `http://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html`

[7] Serialized Lambdas: `http://docs.oracle.com/javase/8/docs/api/java/lang/invoke/SerializedLambda.html`

We extend the ABS API to ABS Remote API[8] that provides the above properties for actors in a seamless way. A complete example of using the remote API has been developed[9]. Expanding our ping-pong example in this paper, Listing 17 and 18 present how a remote server of actors is created for the ping and pong actors. In the following listings, `java.util.Properties` is used provide input parameters of the actor server; namely, the address and the port that the actor server responds to.

Listing 17: Remote ping actor main

```
1  Properties p =  new Properties();
2  p.put("host", "localhost");
3  p.put("port", "7777");
4  ActorServer s = new ActorServer(p)
       ;
5  IPong pong =
6   s.newRemote("abs://pong@http://
        localhost:8888",
7    IPong.class);
8  Ping ping = new Ping(pong);
9  ping.send(
10   () -> ping.ping("")
11 );
```

Listing 18: Remote pong actor main

```
1  Properties p = new Properties();
2  p.put("host", "localhost");
3  p.put("port", "8888");
4  ActorServer s = new ActorServer(p)
       ;
5  Pong pong = new Pong();
```

In Listing 17, a remote reference to a pong actor is created that exposes the `IPong` interface. This interface is proxied [10] by the implementation to handle the remote communication with the actual pong actor in the other actor server. This mechanism hides the communication details from the ping actor and as such allows the ping actor to use the same API to send a message to the pong actor (without even knowing that the pong actor is actually remote). When an actor is initialized in a distributed setting it transparently identifies its actor server and registers with it. The above two listings are aligned with the similar main program presented in Listing 15 that presents the same in a local setting. The above two listings run in separate JVM instances and therefore do not share any objects. In each JVM instance, it is required that both interfaces `IPing` and `IPong` are visible to the classpath; however, the ping actor server only needs to see `Ping` class in its classpath and similarly the pong actor server only needs to see `Pong` class in its classpath.

[8] The implementation is available at `https://github.com/CrispOSS/abs-api-remote`.

[9] An example of ABS Remote API is available at `https://github.com/CrispOSS/abs-api-remote-sample`.

[10] Java Proxy: `http://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html`

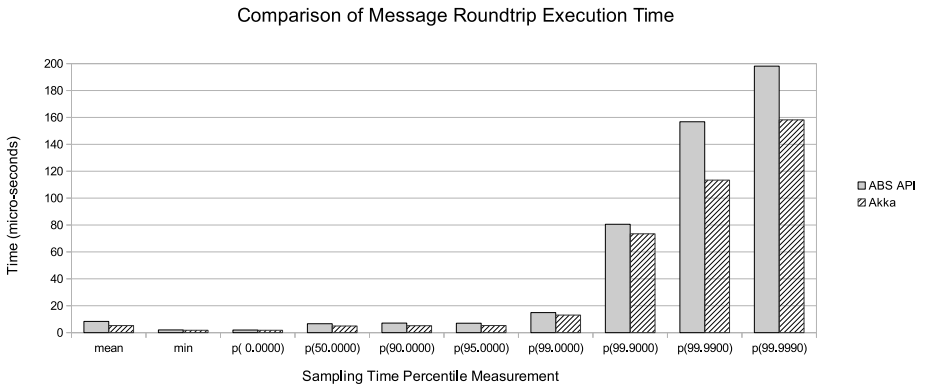

Fig. 2: Benchmark results of comparing sampling time of message round trips in ABS API and Akka. An example reading of above results is that the time shows for $\mathrm{p}(90.0000)$ reads as "message round trips were completed under $10\mu s$ for 90% of the sent messages". The first two columns show the "minimum" and "mean" message round trip times in both implementations.

8 Experiments

In this section, we explain how a series of benchmarks were directed to evaluate the performance and functionality of actor API in Java 8. For this benchmark, we use a simple Java application that uses the "Ping-Pong" actor example discussed previously. An application consists of one instance of `Ping` actor and one instance of `Pong` actor. The application sends a `ping` message to the ping actor and waits for the result. The ping message depends on a `pong` message to the pong actor. When the result from the pong actor is ready, the ping actor completes the message; this completes a round trip of a message in the application. To be able to make comparison of how actor API in Java 8 performs, the example is also implemented using Akka [32] library. The same set of benchmarks are performed in isolation for both of the applications. To perform the benchmarks, we use JMH [30] that is a Java microbenchmarking harness developed by OpenJDK community and used to perform benchmarks for the Java language itself.

The benchmark is performed on the round trip of a message in the application. The benchmark starts with a warm-up phase followed by the running phase. The benchmark composes of a number of iterations in each phase and specific time period for each iteration specified for each phase. Every iteration of the benchmark triggers a new message in the application and waits for the result. The measurement used is *sampling time* of the round trip of a message. A specific number of samples are collected. Based on the samples in different phases, different *percentile* measurements are summarized. An example per-

centile measurement $p(99.9900) = 10\ \mu s$ is read as 99.9900% of messages in the benchmark took 10 micro-seconds to complete.

Each benchmark starts with 500 iterations of warm-up with each iteration for 1 micro-second. Each benchmark runs for 5000 iterations with each iteration for 50 micro-seconds. In each iteration, a maximum number of 50K samples are collected. Each benchmark is executed in an isolated JVM environment with Java 8 version b127. Each benchmark is executed on a hardware with 8 cores of CPU and a maximum memory of 8GB for JVM.

The results are presented in Figure 2. The performance difference observed in the measurements can be explained as follows. An actor in Akka is expected to expose a certain behavior as discussed in Section 3 (i.e. `onReceive`). This means that every message leads to an eventual invocation of this method inside actor. However, in case of an actor in Java 8, there is a need to make a look-up for the actual method to be executed with expected arguments. This means that for every method, although in the presence of caching, there is a need to find the proper method that is expected to be invoked. A constant overhead for the method look-up in order to adhere to the object-oriented principles is naturally to be expected. Thus, this is the minimal performance cost that the actor API in Java 8 pays to support programming to interfaces.

9 Conclusion

In this paper, we discussed an implementation of the actor-based ABS modeling language in Java 8 which supports the basic object-oriented mechanisms and principles of method look-up and programming to interfaces. In the full version of this paper we have developed an operational semantics of Java 8 features including lambda expressions and have proved formally the correctness of the embedding in terms of a bisimulation relation.

The underlying modeling language has an executable semantics and supports a variety of formal analysis techniques, including deadlock and schedulability analysis [10,19]. Further it supports a formal behavioral specification of interfaces [14], to be used as contracts.

We intend to expand this work in different ways. We aim to automatically *generate* ABS models from Java code which follows the ABS design methodology. Model extraction allows industry level applications be abstracted into models and analyzed for different goals such as deadlock analysis and concurrency optimization. This approach of model extraction we believe will greatly enhance industrial uptake of formal methods. We aim to further extend the implementation of API to support different features especially regarding distribution of actors especially in the queue layer, and scheduling of messages using application-level policies or real-time properties of a concurrent system. Furthermore, the current implementation of ABS API in a distributed setting allows for instantiation of remote actors. We intend to use the implementation to model ABS deployment components [21] and simulate a distributed environment.

References

1. G. Agha, I. Mason, S. Smith, and C. Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 7:1–72, 1997.
2. Gul Agha. The Structure and Semantics of Actor Languages. In *Proc. the REX Workshop*, pages 1–59, 1990.
3. Joe Armstrong. Erlang. *Communications of ACM*, 53(9):68–75, 2010.
4. Michael Baker and Martin Thompson. *LMAX Disruptor*. LMAX Exchange. `http://github.com/LMAX-Exchange/disruptor`.
5. Po-Hao Chang and Gul Agha. Towards Context-Aware Web Applications. In *7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*, pages 239–252, 2007.
6. Elaine Cheong, Edward A. Lee, and Yang Zhao. Viptos: a graphical development and simulation environment for tinyOS-based wireless sensor networks. In *Proc. Embedded net. sensor sys., SenSys 2005*, pages 302–302, 2005.
7. Ben Christensen. *RxJava: Reactive Functional Progamming in Java*. Netflix. `http://github.com/Netflix/RxJava/wiki`.
8. Martin Fowler. *LMAX Architecture*. Martin Fowler. `http://martinfowler.com/articles/lmax.html`.
9. Gamma, Erich and Helm, Richard and Johnson, Ralph and Vlissides, John. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *ECOOP '93 – Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 406–431. Springer Berlin Heidelberg, 1993.
10. Elena Giachino, Carlo A. Grazia, Cosimo Laneve, Michael Lienhardt, and Peter Y. H. Wong. Deadlock analysis of concurrent objects: Theory and practice. In *IFM*, pages 394–411, 2013.
11. Brian Goetz. *Lambda Expression Translation in Java 8*. Oracle. `http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html`.
12. Brian Goetz. *Lambda: A Peek Under The Hood*. Oracle, 2012. JAX London.
13. Brian Goetz. *JSR 335, Lambda Expressions for the Java Programming Language*. Oracle, March 2013. `http://jcp.org/en/jsr/detail?id=335`.
14. Reiner Hähnle, Michiel Helvensteijn, Einar Broch Johnsen, Michael Lienhardt, Davide Sangiorgi, Ina Schaefer, and Peter Y. H. Wong. Hats abstract behavioral specification: The architectural view. In *FMCO*, pages 109–132, 2011.
15. Philipp Haller. On the integration of the actor model in mainstream technologies: the Scala perspective. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*, pages 1–6. ACM, 2012.
16. Philipp Haller and Martin Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009.
17. Carl Hewitt. Procedural Embedding of knowledge in Planner. In *Proc. the 2nd International Joint Conference on Artificial Intelligence*, pages 167–184, 1971.
18. Carl Hewitt. What Is Commitment? Physical, Organizational, and Social (Revised). In *Proc. Coordination, Organizations, Institutions, and Norms in Agent Systems II*, LNCS Series, pages 293–307. Springer, 2007.
19. Mohammad Mahdi Jaghoori, Frank S. de Boer, Tom Chothia, and Marjan Sirjani. Schedulability of asynchronous real-time concurrent objects. *J. Log. Algebr. Program.*, 78(5):402–416, 2009.
20. Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects*, pages 142–164. Springer, 2012.

21. Einar Broch Johnsen, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa. Modeling resource-aware virtualized applications for the cloud in Real-Time ABS. In *Formal Methods and Software Engineering*, pages 71–86. Springer, 2012.
22. Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor frameworks for the JVM platform: a comparative analysis. In *Proc. Principles and Practice of Prog. in Java (PPPJ'09)*, pages 11–20. ACM, 2009.
23. Doug Lea. *JSR 166: Concurrency Utilities*. Sun Microsystems, Inc. `http://jcp.org/en/jsr/detail?id=166`.
24. Edward A. Lee, Xiaojun Liu, and Stephen Neuendorffer. Classes and inheritance in actor-oriented design. *ACM Transactions in Embedded Computing Systems*, 8(4), 2009.
25. Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. Actor-Oriented Design of Embedded Hardware and Software Systems. *Journal of Circuits, Systems, and Computers*, 12(3):231–260, 2003.
26. Meyer, B. Applying "design by contract". *Computer*, 25(10):40–51, Oct 1992.
27. Microsoft. *Reactive Extensions*. Microsoft. `https://rx.codeplex.com/`.
28. Behrooz Nobakht, Frank S. de Boer, Mohammad Mahdi Jaghoori, and Rudolf Schlatte. Programming and deployment of active objects with application-level scheduling. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1883–1888. ACM, 2012.
29. John Rose. *JSR 292: Supporting Dynamically Typed Languages on the Java Platform*. Oracle. `http://jcp.org/en/jsr/detail?id=292`.
30. Aleksey Shipilev. *JMH: Java Microbenchmark Harness*. Oracle. `http://openjdk.java.net/projects/code-tools/jmh/`.
31. Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for Java. In *ECOOP 2008–Object-Oriented Programming*, pages 104–128. Springer, 2008.
32. Typesafe. *Akka*. Typesafe. `http://akka.io/`.
33. Jose Valim. *Elixir*. Elixir. `http://elixir-lang.org/`.
34. Carlos A Varela, Gul Agha, Wei-Jen Wang, Travis Desell, Kaoutar El Maghraoui, Jason LaPorte, and Abe Stephens. The SALSA Programming Language 1.1.2 Release Tutorial. *Dept. of Computer Science, RPI, Tech. Rep*, pages 07–12, 2007.
35. Wirfs-Brock, Rebecca J. and Johnson, Ralph E. Surveying Current Research in Object-oriented Design. *Commun. ACM*, 33(9):104–124, 1990.

A.2 ABS: a high-level modeling language for Cloud-Aware Programming

ABS: a high-level modeling language for Cloud-Aware Programming *

Nikolaos Bezirgiannis, Frank de Boer

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

Abstract. Cloud technology has become an invaluable tool to the IT business, because of its attractive economic model. Yet, from the programmers' perspective, the development of cloud applications remains a major challenge. In this paper we introduce a programming language that allows Cloud applications to monitor and control their own deployment. Our language originates from the Abstract Behavioral Specification (ABS) language: a high-level object-oriented language for modeling concurrent systems. We extend the ABS language with Deployment Components which abstract over Virtual Machines of the Cloud and which enable any ABS application to distribute itself among multiple Cloud-machines. ABS models are executed by transforming them to distributed-object Haskell code. As a result, we obtain a Cloud-aware programming language which supports a full development cycle including modeling, resource analysis and code generation.

1 Introduction

The IT industry, always looking for cutting operational costs, has been increasingly relying on virtualized resources offered by the "Cloud". Besides being more economically attractive, the Cloud can allow certain software to benefit in security and execution speed. For these reasons, software applications are steadily being migrated to run on virtualized hardware, essentially turning cloud computing into a hot topic among the software community.

Recent research has led to numerous methodologies, tools, and technologies being proposed to help the migration and execution of software in the cloud, ranging from (static) configuration management tools to (live) orchestration middleware, and from simple resource monitoring services to the dynamic (elastic) provisioning of resources. Unfortunately, the (so-called) DevOps engineers are now burdened with developing and maintaining an extra logic for such cloud tools, besides the usual application logic. These cloud tools may be best described as semi-automatic and it is often the case that an engineer has to manually intervene to apply the desired configuration&deployment of a cloud application.

These cloud applications are migrated unchanged: monolithic boxes of code which are transferred from a non-cloud setting to the new cloud environment by

* Partly funded by the EU project FP7-610582 Envisage. This work was carried out on the Dutch national e-infrastructure with the support of SURF Foundation.

the DevOps engineers. Such separation of the application from its execution is traditionally believed to be an advantage, long before Cloud came to existence. However, one would expect that with the introduction of the virtualized (dynamic) hardware of the Cloud, and since software logic is inherently dynamic, an application could "become aware" and leverage its own execution for managing its cloud resources & deployment in an optimal way, and without the constant administering of an engineer.

In this paper, we aim to address the challenges of engineering cloud applications by introducing a "cloud-aware" programming language that provides certain high-level abstractions for unifying the application logic together with its deployment logic in a single integrated environment, while in the same-time, hiding any lower-level hardware and cloud-provider considerations. The language is intended for DevOps engineers and (potentially) computational scientists who are responsible for both the development and execution of software residing in the Cloud but would rather focus more on the application's logic than manage continuously its deployment. Applications written in the proposed language are christened "cloud-aware" in the sense that they can *actively* monitor and control their own deployment.

The proposed language is based on the *ABstract Specification language* (ABS), a formally-specified, object-oriented modeling language that has been used for both analyzing [1], verifying ([8]), and simulating [5] software programs, as well as running them in production through the various backends developed (currently targeting Java, Erlang, and Haskell). We extend ABS with *Deployment Components* that serve as a suitable abstraction over Cloud Virtual Machines and which allow the application to distribute itself among multiple (provider-agnostic) computing systems. The ABS developer writes code that can dynamically create, monitor and shutdown such Deployment Components (Virtual Machines) and most importantly bring up new objects inside them. To this end, an ABS cloud-application forms a cloud-aware *distributed-object* system, which consists of a number of inter-VM objects that communicate asynchronously, while recording any failures that may happen in the cloud.

An implementation of this extension must be efficient and safe so that it can be put in production code. For this, the Haskell backend of ABS is chosen for translating ABS code to Haskell intermediate code, which is again typechecked and transformed to an executable by an external Haskell compiler. We augment this backend with support for *Cloud-Haskell*, a framework for type-safe, fault-tolerant distributed programming in the Haskell ecosystem. The implementation, although in its infancy, is already being tested in a real cloud environment, exhibiting promising results which are also presented.

2 ABS Language and its Cloud Extension

The ABS (for "ABstract Specification language")[5] is a statically-typed, executable modeling language with formal operational semantics. The language consists of a purely-functional programming core and an imperative, object-

oriented layer. The syntax and behaviour resembles that of Java with two clear differences: side-effectful code cannot be mixed with pure expressions, and class inheritance is abolished in favour of code reuse via delta models[3]. ABS adds, next to the Java-like (passive) objects, builtin support for active (concurrent) objects coupled with cooperative scheduling.

The *functional core* provides a declarative way to describe computation which abstracts from possible imperative implementations of data structures. The primitive types (Int and Rational) can be extended with (possibly recursive) algebraic data types (ADTs) (e.g. `data Bool = True | False`) that can exhibit parametric polymorphism (`List<A>`) and Hindley-Milner type inference. Pure expressions are formed by successive λ-`let` abstractions and applications over values of the defined datatypes (`let x = 3 in x>2 || True`). Function definitions associate a name to a pure expression which is evaluated in the scope where the the expression's free variables are bound to the function's arguments. The functional core supports pattern matching with a **case**-expression which matches a given expression against a list of branches.

The *imperative layer* specifies the interlaced control flow of the concurrent objects in terms of communication, synchronization, and internal computation. This layer extends the functional core (datatype and function definitions) with interface definitions, class definitions, and a main block. Interfaces declare a set of method names to their type-signatures. An interface `extends` other interfaces, in this case inheriting the methods of of its super-interfaces. A class definition declares its (private-only) attributes and a set of interfaces it `implements`. Method implementation bodies are comprised of statements of standard sequential composition $s; s$, assignment $x = rhs$, conditionals, while-loops, and return. Statements can mutate private attributes of the current class, locally-defined variables, and the method's formal parameters. The read-only variable **this** evaluates to the object in which computation occurs. A program's main block is a special method body with no *this* associated object. Classes are *not* types and used only to create object instances that instead are typed-by-interface. Note that interfaces support subtype polymorphism while ensuring strong encapsulation of implementation details.

Methods calls are either synchronous (`v = obj.method(args);`) where the statement is blocked until the method has finished with result v, or asynchronous (`f = obj!method(args);`) where the statement returns immediately with a *future* f (with type `Fut<A>`), without waiting for the method's completion. Each asynchronous method call creates a new *process* which will eventually store the result of the method call into the future reference. The caller can use this future reference to retrieve the result by calling the blocking statement `v = f.get;`. Objects may form a so-called *Concurrent Object Group* (COG), where objects (and their processes) share the same thread of control: at each point in time, only one process of the COG is executing. This process may decide to willfully pass control to another same-group process, by waiting until a future is ready (`await f?;`) or a boolean expression is met (`await exp;`). ABS does not specify

any concrete policy for this cooperative scheduling of processes; it is left to the particular implementation (backend) to decide.

2.1 Extending to the Cloud

We extend the ABS language with syntactic and library support for Deployment Components. A *Deployment Component* (DC), first described in [7], is "an abstraction from the number and speed of the physical processors available to the underlying ABS program by a notion of concurrent resource". Simply put, a DC corresponds to a single (properly-quantified) Virtual Machine which executes ABS code. We restrict the definition of DC to correspond only to a Platform Virtual Machine (VM) residing inside the boundaries of a Cloud infrastructure. Multiple inter-communicating VMs effectively form an ABS cloud application.

To be able to programmatically (at will) create and delete VMs in any language, would require modeling them as first-class citizens of that language. As such, we introduce DCs as first-class citizens to the already-existing language of ABS in the least-intrusive way: by modeling them as objects. All created DC objects are typed by the interface `DC`. Minimal implementation for the methods of the DC interface are `shutdown` for shutting down and releasing the cloud resources of a virtual machine, and `load` for probing its average *system load*, i.e. a metric for how busy the underlying computing-power stays in a period of time. We use the Unix-style convention of returning 3 average values of 1, 5 and 15 minutes. The DC interface resides in the augmented standard library:

```
module StandardLibrary.CloudAPI;
interface DC {
    Unit shutdown();
    Triple<Rat,Rat,Rat> load();
}
```

By having a common DC interface the different cloud backends can agree on a basic service, while still being able to provide additional functionality through sub-interfaces and distinct DC-interfaced classes. Each DC-interfaced class implements the connection to a distinct cloud provider (e.g. Amazon, Openstack). A code skeleton of such a class follows, where the DC (VM) is parameterized by the number of CPU cores and main RAM memory:

```
module StandardLibrary.SomeProvider;

data CpuSpec = Micro | Small | Large;
data MemSpec = GB(Int) | MB(Int);

class SomeProvider(CpuSpec c,MemSpec m) implements DC {
    Unit shutdown() { /*omitted*/ }
    Triple<Rat,Rat,Rat> load() { /*omitted*/ }
}
```

The implementor can expose other properties to DCs, such as, network, number of IO operations, VM region location. A concrete implementation, which is

omitted for brevity, usually involves some high-level ABS logic coupled with low-level code written in a foreign language (in our case Haskell). The average ABS user will not have to provide such connections to the cloud, since we (the implementors) intend to provide class implementations for most major cloud providers/technologies, in an accompanying ABS library. With this approach, we lift the low-level API of the cloud provider (usually XML-RPC) to a *typed* high-level API (e.g. CpuSpec and MemSpec datatypes).

Moving on, we create an object of the SomeProvider class by passing the number of cores and memory measured in GBs as class' formal parameters. The call to "new SomeProvider" contacts the specific cloud provider in the background for bringing up a new VM instance. The provider responds with a unique identifier (commonly the public IP address of the created VM) which is stored in the DC object. Finally, the machine is released by calling `shutdown()`, making the DC object point to `null`.

```
DC dc1 = new SomeProvider(Large, GB(8));
_ future_l1 = dc1 ! load();// underscore infers the type
_ l1 = future_l1.get;
dc1 ! shutdown();
```

The creation of a DC object reference is usually fast, since it involves a single network communication between the current ABS node and the cloud provider. Still, the underlying VM requires considerably more time to boot up and be responsive, depending on factors such as provider's availability, congestion and hardware. To address this, we allow the creation of new dc objects to continue, but we require the program to potentially block when executing the first operation of the newly-created DC, as shown in the example:

```
DC mail_server = new Amazon(..);
DC web_server = new Azure(..);
DC db_server = new Rackspace(..);
mail_server!load(); // will block if DC is not up yet
```

Similar to `this` identifier, a method context contains the **thisDC** read-only variable (with type DC) that points to the VM host of the current executing object. A running ABS node can thus control itself (or any other nodes), by getting its system load or shutting down its own machine. However, after its creation, a running ABS node will remain idle until some objects are created/assigned to it. The **spawns** keyword is added to create objects that "live" and execute in a remote DC:

```
Interf1 o1 = dc1 spawns Cls1(args..);
o1 ! method1(args..);
this.method2(o1);
```

The `spawns` behaves similar to the `new` keyword: it creates a new object (inside a new COG), initializes it, and optionally calls its run method. Indeed, the expression `new Cls1(params)` is equivalent to `thisDC spawns Cls1(params)`. The keyword spawns returns a *remote object reference*, (also called a proxy object; o1 in the above example) that can be called asynchronously for its methods

and passed around as a parameter. Every remote object reference is a single "address" *uniquely identified* across the whole network of nodes. Calls to `spawns` will also (besides `shutdown`, `load`) block a until the VM is up and running. From a theoretical standpoint, a remotely-spawned object must point to the same code (attributes and methods) as in a local object; a remark that is reinforced in the Subsection 3.1.

Whereas the development of ABS code is by-definition provider-dependent — the user has to explicitly specify the class of the cloud provider —, the communication and interaction between the spawned remote objects is (in principle) *provider-agnostic*. To this extent, an ABS user could write an ABS cloud application that spans over multiple cloud providers and, most importantly, different cloud technologies.

Cloud Failures In cloud computing, and in any distributed system in general, failures are more frequent, mostly because of unreliable networks. Based on this fact, we further extend ABS with proper support for extensible, asynchronous exceptions. At the language level, exceptions are pure expressions modeled as single-constructor values of the ADT `Exception`. To define new exceptions the user writes a declaration similar to an ADT declaration, e.g. `exception MyException(Int, List<String>);`. Our cloud extension predefines certain common "local" exceptions (e.g. NullPointerException, DivisionByZeroException) and cloud-related exceptions (e.g. NetworkErrorException, DCAllocationException, DecodingException).

Exception values are either implicitly raised by a primitive operation (e.g. DivisionByZeroException) or explicitly signaled using the `throw` keyword. To recover from exceptions the user writes a `try/catch/finally` block as in Java, the only difference being that the user can pattern-match on each catch-clause for the exception-constructor arguments. Normally, if an exception reaches the outermost caller without being handled, its process will stop. We introduce a special built-in keyword named `die` that changes this behaviour and causes an object to be nullified and *all* of its processes to stop. With this in hand, a distributed application can easily model objects that can be remotely killed:

```
interface Killable { Unit kill(); }
class K implements Killable {  Unit kill() { die; } }
Killable obj = dc1 spawns K();
obj ! kill();
```

Exceptions originating from asynchronous method calls are recorded in the future values and propagated to their callers. When a user calls "future.get;", an exception matching the exception of the callee-process will be raised. If on the other hand, the user does not call "future.get;", the exception will *not* be raised to the caller node. This design choice was a pragmatic one, to allow for fire-and-forget method calls versus method calls requiring confirmation. In our extension, we name this behaviour "lazy remote exceptions", analogous to lazy evaluation strategy.

3 Implementation

For the implementation, we rely on our abs2haskell backend/transcompiler. Haskell is a statically-typed, purely-functional language and, as such, it becomes straightforward to translate the ABS' functional core to Haskell. In the imperative layer, we model interfaces as Haskell's typeclasses, objects as references to mutable data (IORef in the Haskell world), and futures as synchronizing variables (MVar in Haskell). Nominal subtyping is achieved through an upcasting typeclass. An alternative would be to encode OO using extensible records [6], although this method widens the spectrum to structural subtyping.

At runtime, each COG becomes a Haskell lightweight thread (with SMP parallelism). The COG-thread holds a process-enabled *queue*, a process-disabled *table*, and a local *mailbox*. Upon an asynchronous method call, a new process is created and put in the end of the process-enabled queue; note that processes are not threads, they are coroutines (first-class continuations) and thus can be stored as data. The COG resumes the next process from the queue until it reaches an `await` (on a future or a condition), where the process is suspended and moved to the process-disabled table. Later, another process informs the COG (by writing to its mailbox) that the await-condition is met; the COG will move back the process to the enabled queue. This strategy avoids *busy-wait* polling the await conditions of processes.

Moving on to distributed programming, we extend our backend with layered support for Cloud-Haskell[4], a framework for Haskell that replicates Erlang's concurrency & distribution model (message passing) but in a type-safe manner. We reuse the network transports and serialization protocols defined in Cloud Haskell for the ABS transmitted data between Virtual Machines. Each COG-thread is accompanied with a separate Cloud-Haskell thread (also lightweight) that listens for messages in *public* mailbox and *forwards* them to the local mailbox of its associate COG-thread. This approach was chosen to firstly, avoid needless network-serialization between local communication and secondly, treat our distributed extension as optional to our (previously SMP-only) haskell backend.

Serialization ABS data have to be serialized to a standard format before transmitting them between DCs. The serialization of values of primitives and algebraic datatypes are automatically done by Haskell. We serialize object/future references to *proxy references* by serializing their Cloud-Haskell thread ID (network-unique) together with a COG-unique ID, and leaving out their actual attributes/future results. Each asynchronous method call is serialized to a *static closure*, i.e. a static code-pointer to the method (known at compile-time and platform-independent) and a serialized environment of its free variables (method arguments and local variables). No kind of code (source-, byte- or machine-code) corresponding to the method body is transferred. All described-above serializations are type-safe and version-safe, in the sense that we include next to the payload of an ABS datum, its serialized type signature and the library-versions of any types involved; thus, we avoid decoding bugs because of type and library-version mismatches.

Garbage Collection In a local-only setting, all ABS-based values, i.e. ADTs, futures, objects are automatically garbage-collected by the underlying Haskell GC. However, in our distributed setting some object/future references may have to be transmitted outside as proxy references, which results to the local ABS system garbage-collecting "too-early". An obvious solution would be to abolish automatic GC altogether, but that would hinder the development of software applications, especially those supposed to be long-running (as is the norm in cloud applications). On the other hand, introducing *distributed garbage collection* to ABS would allow both local and remote objects to be automatically GC'ed. The downside is that the user can no longer reason about the GC-incurred performance penalty which may be considerable. We chose a middleground where objects are by default GC-enabled and only become disabled when they are remotely communicated over (to another DC). The implementation has been straight-forward: a process appends the local object reference(s) that are transmitted remotely to a locally-held list of GC-disabled objects. This global list is held during the lifetime of the node, effectively surpassing the Haskell's garbage collector underneath. Our design choice was based on best practice; we believe that a distributed cloud ABS application of many DCs would contain a combination of a lot of local ephemeral objects, yet a few long-lived remote objects.

DCs, being special objects, are treated differently: when falling out of context they are automatically GC'ed. That does not mean that the attached VM is shut down. The user that wants to shutdown a DC but holds no reference to it anymore, has to contact a remote object residing there to return a reference to the DC (with `thisDC`), or to shut it down on user's behalf. If the executing program holds (now and in the future) no reference to a DC and its objects, we consider its VM unreachable and fallen out of scope of the ABS application.

Futures are garbage-collected in a publish-subscribe pattern: the caller of an asynchronous method is a subscriber, while the callee is the publisher. When the callee has finished computing the future, it "pushes" the resulted value to its caller (the direct subscriber) and may now locally garbage-collect that value. A subscriber that "passes over" a remote future reference to other nodes becomes an intermediate broker with the responsibility to later also "push" that future value to all others *before* it is allowed to locally garbage-collect it. This forwarding strategy avoids unnecessary tracking and network communication between the initial node and all (directly and indirectly) subscribed nodes.

Cloud Architecture When creating a new DC, a cloud provider is on the background contacted (usually via an XML-RPC API) and asked to bring up a new VM with the given characteristics. After the machine has booted, the caller replicates itself (the current ABS application) by transmitting its machine code to the newly-created machine. In case the cloud provider offers heterogeneous platforms (different OS or CPU architecture), we instead transmit the ABS source code and compile it in-place with our compiler toolset (that prior reside in the VM's image). The new machine runs the transmitted ABS application and sends an acknowledgment signal to its creator, which can now start computations to the new DC by spawning new objects in it.

When it comes to network communication between machines, Cloud-Haskell does not enforce any particular network transport; even different transports can be composed together. Some existing implementations are TCP, AMQP, CCI, in-memory, etc. In ABS, the particular transport used depends on the implementation of the DC-interfaced class: we currently have DC-class implementations for OpenNebula (TCP), Azure (TCP) and Local (in-memory).

4 Experimental Results

We tested two instances of a real-world load-balancer: one with a static deployment of workers, and an adaptive (dynamic) load-balancer with worker VMs created on-demand based on how "well" the workers can keep up with incoming requests. Clients were submitting job requests (of approximately of equal size) to the balancer in a steady rate; workers were distinct Cloud VMs that were continuously computing the results for their incoming job requests.

The *static* load-balancer case is a fairly straight-forward cloud ABS application, consisting of 3 classes of LoadBalancer, Worker, and Client, exchanging asynchronous method calls of job requests/results. The LoadBalancer runs the main block and initially creates N number of Worker DCs (VMs) before starting accepting requests and forwarding to workers in round-robin. We ran this static deployment against varying size (N=1..16) of worker VMs. The results of the runs are shown in Figure 1(a) stripped from the initial boot time of VMs. What we can draw from these results is that the completed jobs (per minute) nearly doubles when we double the number of worker VMs until we reach 5 workers. After that, we still increase the completed jobs but with a slower pace. This observation can be attributed to the fact that a point is reached where there is not a significant benefit from adding more worker VMs; the rate of job requests is always steady, thus worker VMs are "slacking".

We modified the static load-balancer to an *adaptive* version, that takes full advantage of the expressivivity of the cloud extension. The LoadBalancer creates now only 1 initial VM. We accommodate the LoadBalancer with a HeartBeater object which periodically retrieves the load of each worker in the VM "farm". The HeartBeater computes the average `load` of all VMs and if this average exceeds 80%, it creates a new DC (VM), adds it to the current farm, and remotely spawns a Worker in the new DC. We illustrate a particular run of this configuration in Figure 1(b) (NB: VM boot times are not subtracted from the result). Each asterisk $*$ in (b) is a point where the HeartBeater decides to create a new DC. This run stabilizes on 6 workers, which is a good approximation of maximum speed (according to Figure 1(a)), and possibly a good choice if we took into account any VM costs. As an extra, the HeartBeater could potentially `shutdown` machines if their load remained small (under a threshold) for a certain time.

The tests were conducted on the SURF cloud-provider with OpenNebula IaaS, modern 8-cores, each with 8GB RAM and 20Gbps Ethernet. Interesting to

mention is that each worker can benefit from ABS multicore (SMP) parallelism. A snippet of the HeartBeater follows with the full ABS code at our repository[1]:

```
class HeartBeater(List<Worker> farm, Balancer b) {
  Unit beat() {
    Rat avg = this.computeLoads(farm);
    if (avg > 80/100) {
      DC dc = new NebulaDC(8,8192); // 8-core, 8GB RAM
      Worker w = dc spawns Worker();
      farm = Cons(w,farm);
      b ! updateFarm(farm); } } }
```

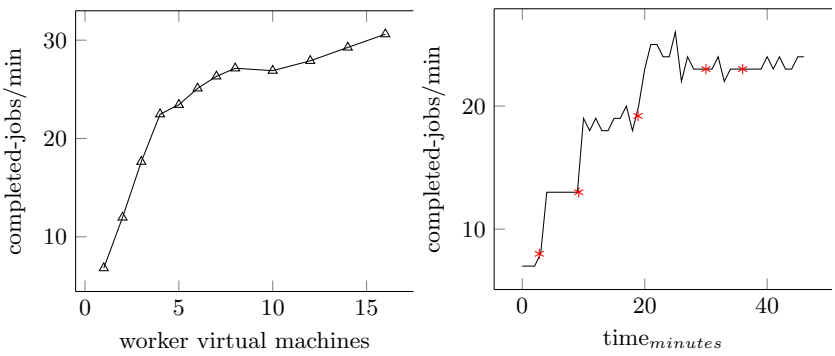

Fig. 1: (a) Static deployment of VMs (b) Adaptive Deployment over time

5 Related work

With the introduction of the Cloud, a plethora of cloud technologies & tools have appeared in the software community. We distinguish two categories of technologies related to our work: distributed-prog. languages and cloud middleware.

Distributed languages Erlang is one of the first distributed-oriented languages that next to the canonical message-passing communication, offers distinct features, such as hot-code loading and serialization of arbitrary closures. This comes with a cost in safety since the serialized Erlang data are untyped and usually unversioned. Erlang's builtin processes are lightweight threads whereas ABS processes are coroutines (even more lightweight). The Akka framework brings (typed) actors to the Scala language. Although Akka provides a rich library and toolkit, it currently lacks a cloud-aware API. At runtime Akka is constrained by a threadpool (since JVM threads are expensive) and actors are not able to use

[1] Upstream abs2haskell repository at http://github.com/bezirg/abs2haskell

cooperative scheduling and instead resort to a form of message routing. The Java RMI (Remote Method Invocation) is a library bundled in the Java platform for communication between remote objects. The product pioneered in areas such as bytecode downloading and distributed-GC. The method invocation is strictly synchronous (the caller has to wait for the remote method to finish) and thread-unsafe. JADE[2] is an active distributed multiagent system also built in Java; agents are more expressive than actors at the expense of program complexity and, possibly, performance.

Cloud middleware Ubuntu JuJu is a tool primarily for scaling and orchestrating a system's deployment on the cloud. Juju also comes with a GUI for modeling and visualizing a cloud deployment and saving it to a "recipe" for later reuse. It is usually accompanied by a configuration-management tool (such as Puppet) for the provisioning of cloud machines. CoreOS is a container-based OS that provides service and configuration discovery. It can be thought as a low-level infrastructure, primarily targeted to system administrators, for managing system services across a cluster of cloud machines, The Aeolus research project has built various tools that can derive an optimized deployment from the constraint-based model of a desired deployment, and automatically deploy that derivation. Finally, general SaaS supported by cloud providers eases the migration of existing software to the cloud and its automatic scaling of deployment. Albeit dynamic, a SaaS deployment can only vary on the CPU consumption, whereas our proposal would allow a much more expressive deployment that can depend on arbitrary application logic.

6 Conclusion and Future Work

We presented an extension to the ABS language that permits the management of an application's own cloud-deployment inside the language itself. We discussed the realization of such extension (by a Haskell transcompiler) and the execution of an ABS cloud application (based on Cloud-Haskell). Results showed that ABS can benefit from the extra performance that the Cloud offers. Moreover, the extension gives to ABS the expression power it needs to fuse the application logic with the application's own (dynamic) deployment logic. A positive side-effect of the proposed extension is that, ABS being primarily a modeling language, could now be used to model also an application's deployment. Indeed, such cloud-aware software models could be simulated against different and dynamically-varying cloud deployment scenarios.

For future work we are considering additions both at the language and run-time level. At the language level, it would be beneficial to include, besides the system load, other metrics such as memory, disk usage, object count, process count, exceptions raised. In this way, an ABS application would enhance its monitor and cloud-control logic. In a different direction, we plan to work on adding a basic *service discovery* mechanism to the standard library of ABS. This can be simply realized by extending the DC interface with two extra methods: an `acquire(Interface obj)` method that returns a reference to a remote

object implementing the provided Interface; an `expose(Interface obj)` that subscribes the passed object together with its current interface-view to the service registry of the DC.

At the system level, we are first interested in expanding our library support for other common cloud providers (such as Amazon EC2, OpenStack). Besides the current open (peer-to-peer) topology of DCs we want to add support for other cloud topologies, such as provider-specific, slave-master, or supervision topologies – a crude solution to topologies would be to introduce to the DC interface a method `List<DC> neighbours()` that lists all ABS nodes residing in the same private cloud network. A second consideration is to extend our virtualization technology support. With the introduction of micro-kernels (see the Xen hypervisor and unikernels), the cloud user no longer needs an OS underneath the application/service. By packaging the application into the kernel itself, the startup time of the VM is greatly improved, as is its management & distribution. The Haskell Lightweight Virtual Machine (HaLVM) is a promising such technology that allows the user to: "run Haskell programs without a host operating system". Likewise, *containers* (e.g. Docker), with its OS-level virtualization, would allow us to offer a more fine-grained control of deployment.

We believe that the cloud extension of ABS leads to new opportunities for furthering the application of formal methods to cloud computing, for example: specifying, verifying, and monitoring Service Level Agreements (SLA) of software systems — with that being the overall goal of ENVISAGE, our current research project. Indeed, we like to envisage software that is aware of its deployment and thus can control it, while its users merely monitor its behaviour via SLAs signed between the interested parties.

References

1. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: Costabs: a cost and termination analyzer for abs. In: PEPM. pp. 151–154 (2012)
2. Bellifemine, F., Poggi, A., Rimassa, G.: Jade–a fipa-compliant agent framework. In: Proceedings of PAAM. vol. 99, p. 33. London (1999)
3. Clarke, D., Helvensteijn, M., Schaefer, I.: Abstract delta modeling. ACM Sigplan Notices 46(2), 13–22 (2011)
4. Epstein, J., Black, A.P., Peyton-Jones, S.: Towards haskell in the cloud. In: ACM SIGPLAN Notices. vol. 46. ACM (2011)
5. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: Abs: A core language for abstract behavioral specification. In: FMCO. pp. 142–164 (2010)
6. Kiselyov, O., Lmmel, R., Schupke, K.: Strongly typed heterogeneous collections. In: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell. pp. 96–107 (2004)
7. Schäfer, J., Poetzsch-Heffter, A.: Jcobox: Generalizing active objects to concurrent components. In: ECOOP 2010–Object-Oriented Programming. Springer (2010)
8. Wong, P.Y.H., Albert, E., Muschevici, R., Proença, J., Schäfer, J., Schlatte, R.: The abs tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. STTT 14(5), 567–588 (2012)

A.3 A Formal, Resource Consumption-Preserving Translation of Actors to Haskell

A Formal, Resource Consumption-Preserving, and Efficient Translation of Actors to Haskell*

Elvira Albert[1], Nikolaos Bezirgiannis[2], Frank de Boer[2], and Enrique Martin-Martin[1]

[1] Universidad Complutense de Madrid, Spain
elvira@sip.ucm.es, emartinm@ucm.es
[2] Centrum Wiskunde & Informatica (CWI), Amsterdam, Netherlands
{n.bezirgiannis, f.s.de.boer}@cwi.nl

Abstract. In this paper we present and discuss a formal translation of a concurrent actor-based language which supports *cooperative scheduling* into the functional language Haskell. The translation is proven correct with respect to a formal semantics of the actor-based language and a high-level operational semantics of a subset of the Haskell language that is used as the target language. The main correctness theorem is expressed in terms of a simulation relation between the operational semantics of actor-based programs and their translation. We further prove the preservation of the *resource-consumption* through the translation by establishing the equivalence between the cost of executions in the original and in the Haskell-translated program.

1 Introduction

Abstract Behavioural Specification Language (ABS) [9] is a formally-defined, *actor-based*, executable modeling language which supports various analysis and verification tools [15]. Actor programs consist of computing entities called actors, each with its own local state and thread of control, that communicate by exchanging messages asynchronously. In ABS, the notion of actor corresponds to that of *concurrent object*, where objects are the concurrency units, i.e., each object conceptually has a dedicated processor. Communication is based on asynchronous method calls with standard objects as targets such that method calls trigger potentially concurrent processes. Concurrent objects execute activities (processes) from their process queue. ABS supports *cooperative scheduling*, which means that inside the object's monitor an active process can decide to explicitly suspend its execution so as to allow another process from the queue to execute. This way, the interleaving of processes inside a concurrent object is textually controlled by the programmer. However, flexible and state-dependent interleaving is still supported; in particular, a process may suspend its execution waiting for a reply to a method call. We use *future variables* [8,5] to check if the execution of an asynchronous task has finished.

* Partly funded by the EU project FP7-610582 Envisage, by the Spanish MINECO project TIN2012-38137, and by the CM project S2013/ICE-3006.

The overall contribution of this paper is a formal, resource-consumption preserving, and efficient transformation of the core subset of the ABS language into Haskell. The transformation consists in compiling ABS methods into Haskell functions with continuations (similar transformations have been performed in the actor-based Erlang language wrt. *rewriting systems* [13,14] and *rewriting logic* [12], and in the translation of ABS to Prolog [3] and an ABS subset to Scala [10]). However, what it is unique in our transformation and constitutes our main contribution is:

- *Soundness.* We provide a formal statement of the soundness of this translation of ABS into Haskell which is expressed in terms of a simulation relation between the operational ABS semantics and the semantics of the generated Haskell code. The soundness claim ensures that every Haskell derivation has an equivalent one in the ABS code. However, since for efficiency reasons, the translation fixes a selection order between the objects and the tasks within each object, we do not have a completeness result.
- *Resource-preservation.* We prove formally that the transformation preserves the resource consumption, i.e., given a cost model that assigns a cost to each instruction of the source language, we prove that the cost of executing the Haskell-translated program is the same as executing the original ABS program since both execute the same instructions. Having this result allows us to ensure that upper bounds on the resource consumption obtained by the analysis of the original ABS program are preserved during compilation and are thus valid bounds for the Haskell-translated program as well.
- *Efficiency.* We experimentally assess the efficiency of our transformation using the canonical, full ABS-to-Haskell compiler, and an extract of it where we base our formalizations on. Using the two systems, we carry out two types of experiments. (1) First, we illustrate that the compilation does not introduce an overhead during execution: run-times are proportional to the number of steps executed. For this, we use the extracted compiler and several micro-benchmarks that feature different complexity bounds and we compute the number of executed steps, the execution time and the upper bounds on the number of executed steps given by static analysis. In all cases, our experiments confirm the proportionality of steps and time, and also that the upper bounds obtained for the ABS program are an accurate estimate of the actual consumption. (2) We compare the efficiency of our full-blown ABS-to-Haskell compiler w.r.t. previous compilers from ABS to different languages (namely Erlang, Java and Maude) and prove experimentally that our compiler outperforms the previous ones both in time and memory consumption.

2 Source language

Our language derives from ABS [9], a statically-typed, actor-based language with a purely-functional core (ADTs, functions, parametric polymorphism) and an object-based imperative layer: objects with private-only attributes, and interfaces that serve as types to the objects. ABS extends the OO paradigm with

$$
\begin{array}{lll}
S ::= & x := E \mid f := x!m(\bar{y}) & \\
& \mid \texttt{await}\ f \mid \texttt{skip} \mid \texttt{return}\ z & \\
& \mid S_1; S_2 \mid \texttt{if}\ B\ \{S\}\ \texttt{else}\ \{S\} & \\
& \mid \texttt{while}\ B\ \{S\} & \\
E ::= & x \mid r \mid \texttt{new} \mid f.\texttt{get} \mid m(\bar{y}) & \\
D ::= & m(\bar{r}) \mapsto S & \\
P ::= & \overline{D}\ :\ \texttt{main}() \mapsto S &
\end{array}
$$

```
1  main() ->
2    x = new;
3    y = new;
4    f1 = x ! task1();
5    f2 = y ! task2();
6    await f1;
7    r1 = f2.get;
8    r2 = task3(r1);
9    return r2;

10 task1() -> ...
11 task2() -> ...
12 task3(r1) -> ...
```

Fig. 1: (a) syntax of source language (b) a running ABS example

support for *asynchronous* method calls; each call results to a new *future* (placeholder for the method's result) returned to the caller-object, and a new process (stored in the callee-object's process queue) which runs the method's activation. The active process inside an object (only one at any given time) may decide to explicitly suspend its execution so as to allow another process from the same queue to execute.

In this paper, we deal with a subset of ABS concerning the concurrent interaction of processes both inside and between objects; in a nutshell, the language is stripped of its functional core and all types/interfaces. The choice was made to focus strictly on the more challenging part of proving correctness of the cooperative concurrency. However, the full-blown compiler that we will use in the experiments deals with the whole ABS language. The formal syntax of the statements S of the subset is shown in Fig. 1(a). Values in our subset are object and future references (disjoint); values can be stored in method's formal parameters or attributes. We syntactically distinguish between method parameters r and attributes. The attributes are further distinguished for the values they hold: object attributes holding object references (denoted by $x, y, z \ldots$), and future attributes holding future references (denoted by f). An assignment $f := x!m(\bar{y})$ stores to the future attribute f a new future reference returned by asynchronously calling the method m on the object attribute x passing as arguments the values of object attributes $\bar{y}$. An assignment $x := E$ stores to an object attribute the result of executing the right-hand side E. A right-hand side can be the value of a method parameter r or an attribute x, a reference to a new object `new`, the result of a synchronous same-object method call $m(\bar{y})$, or the result of an asynchronous method call $f.get$ stored in the future attribute f. A call to $f.get$ will block the object and all its processes until the result of the asynchronous call is ready. The statement `await` f may be used (usually before calling $f.get$) to instead release the current process until the result of f has been computed, allowing another same-object process to execute. Sequential composition of two statements S_1 and S_2 is denoted by $S_1; S_2$. The Boolean condition B in the *if* and *while* statement is a Boolean combination of reference equality between values of attributes. The statement `return` z returns the value of the attribute z both in synchronous and asynchronous method calls. A method declaration D maps a method's name and formal parameters to a statement S (method body). We consider that every method has one `return` and it is the final statement.

$$(\textsc{Assign I})\ \frac{h' = h[(n)(x) \mapsto h(n)(y)]}{\langle n : (\texttt{x:=y}; S, l) \cdot Q, h\rangle \to \langle n : (S, l) \cdot Q, h'\rangle}$$

$$(\textsc{Assign II})\ \frac{h' = h[(n)(x) \mapsto r]}{\langle n : (\texttt{x:=r}; S, l) \cdot Q, h\rangle \to \langle n : (S, l) \cdot Q, h'\rangle}$$

$$(\textsc{New})\ \frac{h(\mathit{count}) = m \qquad h' = h[(n)(x) \mapsto m, (m) \mapsto \epsilon, \mathit{count} \mapsto m + 1]}{\langle n : (\texttt{x:=new}; S, l) \cdot Q, h\rangle \to \langle n : (S, l) \cdot Q, h'\rangle}$$

$$(\textsc{Get})\ \frac{h(h(n)(f)) \neq \bot \qquad h' = h[(n)(x) \mapsto h(h(n)(f))]}{\langle n : (\texttt{x:=f.get}; S, l) \cdot Q, h\rangle \to \langle n : (S, l) \cdot Q, h'\rangle}$$

$$(\textsc{Await I})\ \frac{h(h(n)(f)) \neq \bot}{\langle n : (\texttt{await f}; S, l) \cdot Q, h\rangle \to \langle n : (S, l) \cdot Q, h\rangle}$$

$$(\textsc{Await II})\ \frac{h(h(n)(f)) = \bot}{\langle n : (\texttt{await f}; S, l) \cdot Q, h\rangle \to \langle n : Q \cdot (\texttt{await f}; S, l), h\rangle}$$

$$(\textsc{Async})\ \frac{\begin{array}{c} h(n)(x) = d \qquad h(\mathit{count}) = l \qquad \bar{r} = h(n)(\bar{z}) \\ h' = h[(n)(f) \mapsto l, (l) \mapsto \bot, \mathit{count} \mapsto l + 1] \end{array}}{\langle n : (\texttt{f:=x!m}(\bar{z}); S, l) \cdot Q, h\rangle \overset{d.m(l,\bar{r})}{\longrightarrow} \langle n : (S, l) \cdot Q, h'\rangle}$$

$$(\textsc{Sync})\ \frac{(m(\bar{w}) \mapsto S_m) \in D \ \mathit{fresh} \qquad \tau = [\bar{w} \mapsto h(n)(\bar{z})] \qquad S' = \widehat{(S_m\tau)}^{\,x}}{\langle n : (\texttt{x:=m}(\bar{z}); S, l) \cdot Q, h\rangle \to \langle n : (S'; S, l) \cdot Q, h\rangle}$$

$$(\textsc{Return}_A)\ \frac{h' = h[(l) \mapsto h(n)(x)]}{\langle n : (\texttt{return}^{*}\texttt{x}; S, l) \cdot Q, h\rangle \to \langle n : Q, h'\rangle}$$

$$(\textsc{Return}_S)\ \frac{h' = h[(n)(z) \mapsto h(n)(x)]}{\langle n : (\texttt{return}^{z}\ \texttt{x}; S, l) \cdot Q, h\rangle \to \langle n : (S, l) \cdot Q, h'\rangle}$$

Fig. 2: Operational semantics: Local rules

Finally, a program P is a set of method declarations $\bar{D}$ and a special method `main` that has no formal parameters and acts as the program's entry point.

The program of Fig. 1(b) exemplifies the creation of object `x` (line L2) and `y` (L3), and the asynchronous method calls of `task1` in `x` (L4) and `task2` in `y` (L5). In L6 the `await` statement will wait for the termination of `task1`, releasing the processor so that any other process in the same object of `main` can execute. On the other hand, the `get` statement in L7 will wait for the result of `task2` by blocking the object (i.e. main) and all of its processes until the result is ready. Finally, L8 contains a synchronous-method self call to `task3`.

2.1 Operational semantics

In order to describe the operational semantics of the language defined above we first introduce the following concepts and assumptions. We use the set *Nat* of natural numbers to encode object and future references, used to identify dynamically generated objects and futures. We denote by $\Sigma = \mathit{IVar} \to \mathit{Nat}$ the set of assignments of references, presented by natural numbers, to the instance variables (of an object), with typical element σ and empty element ϵ. A closure

consists of a statement S obtained by replacing its free local variables by actual references (note that local variables are introduced as method parameters and can only appear in E) and a future reference, represented by a number, for storing the return value. By $S\tau$, where $\tau \in LVar \to Nat$, we denote the instantiation obtained from S by replacing each local variable x in S by $\tau(x)$. Finally, we represent the global heap h by a triple (n, h_1, h_2) consisting of a natural number n and *partial* functions (with finite disjoint domains) $h_1 : Nat \to \Sigma$ and $h_2 : Nat \to Nat_\perp$, where $Nat_\perp = Nat \cup \{\perp\}$ ($\perp$ is used to denote "undefined"). The number n is used to generate references to new objects and futures. The function h_1 specifies for each existing object, i.e., a number n such $h_1(n)$ is defined, its *local* state. The function h_2 specifies for each existing future reference, i.e., a number n such $h_2(n)$ is defined, its return value (absence of which is indicated by $\perp$). In the sequel we will simply denote the first component of h by $h(count)$, and write $h(n)(x)$, instead of $h_1(n)(x)$, and $h(n)$, instead of $h_2(n)$. We will use the notation $h[count \mapsto n]$ to generate a heap equal to h but with the counter set to n. A similar notation $h[n \mapsto \perp]$ will be used for future variables, $h[(n)(x) \mapsto v]$ for storing the value v in the variable x in object n and $h[n \mapsto \epsilon]$ for initializing the mapping of an object.

An object's *local* configuration denoted by the (object) reference n consists of a pair $\langle n : Q, h\rangle$ where Q is a list of closures and h is the global heap. We use $\cdot$ to concatenate lists, i.e., $(S, l) \cdot Q$ represents a list where (S, l) is the head and Q is the tail. A *global* configuration—denoted with the letters A and B—is a pair $\langle C, h\rangle$ containing a set of lists of closures $C = \{\overline{Q}\}$ and a global heap h. Fig. 2 contains the relation that describes the local behavior of an object (omitting the standard rules for sequential composition, choice and iteration statements). Note that the first closure of the list Q is the active process of the object, so the the different rules process the first statement of this closure. When the active process finishes or releases the object in an `await` statement, the next process in the list will become active, following a FIFO policy. The rules (Assign I) and (Assign II) modify the heap storing the new value of variable x of object n. The (New) rule stores a new object reference in variable x, increments the counter of objects references and inserts an empty mapping ϵ for the variables of the new object m. Rule (Get) can only be applied if the future is available, i.e., if its value is not $\perp$. In that case, the value of the future is stored in the variable x. Both rules (Await I) and (Await II) deal with `await` statements. If the future f is available, it continues with the same process. Otherwise it moves the current process to the end of the queue, enforcing a FIFO policy. Note that the `await` statement is not consumed, as it must be checked when the process becomes active again. When invoking the method m asynchronously in rule (Async) the destination object d and the values of the parameters $\bar{r}$ are computed. Then a new future reference l initialized to $\perp$ is stored in the variable f, and the counter is incremented. The information about the new process that must be created is included as the decoration $d.m(l, \bar{r})$ of the step. Synchronous calls—rule (Sync)—extend the active task with the statements of the method body, where the parameters have been replaced by their value using the substitution

$$(\textsc{Internal})\frac{\langle n : Q, h\rangle \to \langle n : Q', h'\rangle}{\langle (n : Q) \cup C, h\rangle \to \langle (n : Q') \cup C, h'\rangle}$$

$$(\textsc{Message})\frac{\langle n : Q_n, h\rangle \xrightarrow{d.m(l,\bar{r})} \langle n : Q', h'\rangle \quad m(\bar{w}) \mapsto S_m \in D \quad \tau = [\bar{w} \mapsto \bar{r}] \quad S' = \widehat{(S_m \tau)}^*}{\langle (n : Q_n) \cup (d : Q_d) \cup C, h\rangle \to \langle (n : Q') \cup (d : Q_d \cdot (S', l)) \cup C, h'\rangle}$$

Fig. 3: Operational semantics: Global rules

τ. In order to return the value of the method and store it in the variable x, the `return` statement of the body is marked with the destination variable x, called *write-back variable*. This marking is formalized in the $\widehat{\cdot}^x$ function, defined as follows (recall that `return` is the last statement of any method):

$$\widehat{S}^x = \begin{cases} S_1; \widehat{S_2}^x & \text{if } S = S_1; S_2, \\ \texttt{return}^x\ \texttt{z} & \text{if } S = \texttt{return z}, \\ S & \text{i.o.c.} \end{cases}$$

Rule ($\textsc{Return}_A$) finishes an asynchronous method invocation (in this case the `return` keyword is marked with *, see rule (Message) in Fig. 3), so it removes the current process and stores the final value in the future l. On the other hand, rule ($\textsc{Return}_S$) finishes a synchronous method invocation (marked with the write-back variable), so it behaves like a `z:=x` statement.

Based on the previous rules, Fig. 3 shows the relation describing the global behavior of configurations. The (Internal) rule applies any of the rules in Fig. 2, except (Async), in any of the objects. The (Message) rule applies the rule (Async) in any of the objects. It creates a new closure $(\widehat{S_m \tau}^*, l)$ for the new process invoking the method m, and inserts it at the back of the list of the destination object d. Note the use of $\widehat{\cdot}^*$ to mark that the `return` statement corresponds to an asynchronous invocation. Note that in both (Internal) and (Message) rules the selection of the object to execute is non-deterministic. In parts of the proof we decorate both local and global steps with object reference n and statement S executed, i.e., $\langle n : Q, h\rangle \to^n_S \langle n : Q', h'\rangle$ and $\langle C, h\rangle \to^n_S \langle C', h'\rangle$.

We remark that the operational semantics shown in Fig. 2 and 3 is equivalent to the ABS semantics presented in [9], considering that every object is a *concurrent object group*. The main difference is the representation of configurations: in [9] configurations are sets of futures and objects that contain their local stores, whereas in our semantics all the local stores and futures are merged in a global heap. Finally, our operational semantics considers a FIFO policy in the processes of an object, whereas [9] left the scheduling policy unspecified.

3 Target language

Our subset of ABS is compiled to Haskell, with the compiler itself written in Haskell. The Haskell language (statically-typed, purely-functional) enabled us to embed (implement) our language in such a way that the end programs have

```
data Stm = Skip Stm                                  data Rhs = New
     | Assign Attr Rhs Stm                                | Get Attr
     | Await Attr Stm                                     | Async Attr Method [Attr]
     | If BExp (Stm→Stm) (Stm→Stm) Stm                    | Sync Method [Attr]
     | While BExp (Stm→Stm) Stm                           | A Attr
     | Return Attr (Maybe Ref) Stm                       | R Ref

data BExp = BExp :|| BExp | BExp :&& BExp | Not BExp | Attr :== Attr
where  type Ref=Int; type Attr=Int
       type Method=[Ref]→Ref→Maybe Ref→Stm→Stm
```

Fig. 4: The syntax and types of the target language

reasonable execution speed. Moreover, thanks to our soundness result (see Sec. 4) we can transfer the results of ABS analysis tools to the Haskell translated code.

The abstract syntax of this embedded language is shown in Fig. 4. The values of our language are object and future references (named `Ref`), represented by integer indices to the program's global heap array. Similarly, an object attribute `Attr` is an integer index to an internal-to-the-object attribute array, hence shallow-embedded (compared to embedding the actual name of the attribute). In contrast, all statements are deep-embedded as a recursive datatype `Stm`: the recursive position at the end of *each* statement holds the current continuation after the execution of that statement. The body of `While` and the two branch bodies of `If` are given in the higher-order abstract syntax `Stm → Stm`; this function is in continuation-passing style (CPS) for "tying" the body's last statement to the continuation following after that control structure. A `Method` definition is a CPS function which takes as input a list `[Ref]` of the method's arguments (passed by reference), the callee object named `this`, a *writeback* reference (`Maybe Ref`), a continuation `Stm` and returns the method's body as a `Stm`. Upon executing `Return` and in case of synchronous call, the callee method writes the return value to the writeback reference and the execution jumps back to the caller by invoking the method's continuation; in case of asynchronous call the writeback is empty, the return value is stored to the caller's future (destiny) and the method's continuation is invoked resulting to the exit of the ABS process.

The right-hand side (`Rhs` in Fig. 4) of an assignment statement directly reflects that of the source language, with the exception of the `A` and `R` constructors to disambiguate respectively between attributes that must be dereferenced from the object's attribute array, or values bound to formal parameters and thus treated as is. Boolean expressions are only appearing as arguments to `If` and `While` and are inductively constructed through the datatype `BExp`, which represents reference equality between the attributes' values.

The compilation of statements is shown in Fig. 5. The translation ${}^{s}[\![S]\!]_{k,wb}$ takes two arguments: the continuation k and the writeback reference wb. Each statement is translated into its Haskell counterpart, followed by the continuation k. The multiple rules for the `return` statement are due to the different uses of the translation: when compiling methods the `return` statement will appear unmarked, so we include the writeback passed as an argument; otherwise it

$$
\begin{aligned}
{}^{s}[\![\texttt{x:=y}]\!]_{k,wb} &= \texttt{Assign}\ x\ (\texttt{Attr}\ y)\ k & {}^{s}[\![\texttt{skip}]\!]_{k,wb} &= \texttt{Skip}\ k \\
{}^{s}[\![\texttt{x:=r}]\!]_{k,wb} &= \texttt{Assign}\ x\ (\texttt{Param}\ r)\ k & {}^{s}[\![\texttt{await f}]\!]_{k,wb} &= \texttt{Await}\ f\ k \\
{}^{s}[\![\texttt{x:=new}]\!]_{k,wb} &= \texttt{Assign}\ x\ \texttt{New}\ k & {}^{s}[\![\texttt{return x}]\!]_{k,wb} &= \texttt{Return}\ x\ wb\ k \\
{}^{s}[\![\texttt{x:=f.get}]\!]_{k,wb} &= \texttt{Assign}\ x\ (\texttt{Get}\ f)\ k & {}^{s}[\![\texttt{return}^{*}\ \texttt{x}]\!]_{k,wb} &= \texttt{Return}\ x\ \texttt{Nothing}\ k \\
{}^{s}[\![\texttt{x:=y!m}(\bar{z})]\!]_{k,wb} &= \texttt{Assign}\ x\ (\texttt{Async}\ y\ m\ \bar{z})\ k & {}^{s}[\![\texttt{return}^{z}\ \texttt{x}]\!]_{k,wb} &= \texttt{Return}\ x\ (\texttt{Just}\ z)\ k \\
{}^{s}[\![\texttt{x:=m}(\bar{z})]\!]_{k,wb} &= \texttt{Assign}\ x\ (\texttt{Sync}\ m\ \bar{z})\ k & {}^{s}[\![S_1; S_2]\!]_{k,wb} &= {}^{s}[\![S_1]\!]_{k',wb} \text{ with } k' = {}^{s}[\![S_2]\!]_{k,wb}
\end{aligned}
$$

$$
{}^{s}[\![\texttt{if}\ B\ \{S_1\}\ \texttt{else}\ \{S_2\}]\!]_{k,wb} = \texttt{If}\ {}^{B}[\![B]\!]\ (\backslash k' \to {}^{s}[\![S_1]\!]_{k',wb})\ (\backslash k' \to {}^{s}[\![S_2]\!]_{k',wb})\ k
$$

$$
{}^{s}[\![\texttt{while}\ B\ \{S\}]\!]_{k,wb} = \texttt{While}\ {}^{B}[\![B]\!]\ (\backslash k' \to {}^{s}[\![S]\!]_{k',wb})\ k
$$

$$
{}^{m}[\![m]\!] = \texttt{m l this wb k} = {}^{s}[\![S_m]\!]_{\texttt{k,wb}}
$$

where $m(\bar{w}) \mapsto S_m \in D$ and `l` is the Haskell list that contains the same elements as the sequence $\bar{w}$

Fig. 5: Compilation of programs

```
main [] this wb k =
  Assign x New $
  Assign y New $
  Assign f1 (Async x task1 []) $
  Assign f2 (Async y task2 []) $
  Await f1 $
  Assign r1 (Get f2) $
  Assign r2 (Sync task3 [r1]) $
  Return r2 wb k

task1 [] this wb k = ...
task2 [] this wb k = ...
task3 [r3] this wb k = ...

-- Position in the attribute array
[x,y,f1,f2,r1,r2] = [0..]
-- method definitions have type Method
main,task1,task2,task3 :: Method
```

Fig. 6: Compiled running example

is used to translate runtime configurations, so `return` statements will appear marked and we generate the writeback related to the mark. When omitted, we assume the default values $k =$ `undefined` and $wb =$ `Nothing` for the ${}^{s}[\![S]\!]_{k,wb}$ translation. ${}^{B}[\![B]\!]$ represents the translation of a boolean expression B. A method definition translates to a Haskell function that includes the compiled body.

Runtime. The program heap is implemented as the triple: array of objects, array of futures and a `Int` counter. Every cell in the objects array designates a single object holding a pair of its attribute array (implemented in Haskell as `IOVector (IOVector Ref, Seq Proc)`) and process queue (double-ended). A cell in the futures array denotes a future which is either unresolved with a number of listener-objects `await`ing for it to be completed, or resolved with a final value, hence the type `IOVector (Either [Ref] Ref)`. An ever-increasing counter is used to pick new references; when it reaches the arrays' current size both of the arrays double in size (i.e. dynamic arrays). The size of all attribute arrays, however, is fixed and predetermined at compile-time; we assume a prior static analysis that maps attribute names to indexes for the attribute array.

The function `eval` accepts a `this` object reference and the current heap and executes a single statement of the head process in the process queue, returning a new heap and those objects that have become active after the execution (`eval this heap :: IO (Heap, [Ref])`). An `await` executed statement will put its continuation (current process) in the tail of the process queue, ef-

fectively enabling cooperative multitasking, whereas all others will keep it as the head. A `Return` executed statement originating from an asynchronous call is responsible for re-activating the objects that are blocked on its resolved future. A global scheduler (`sched` function) keeps a queue of active objects; it calls `eval` on the head object, puts the newly activated objects in the tail of the queue, and loops until no objects are left in the object queue, and then it exits (the ABS program is finished or deadlocked). At any point in time, the pair of the scheduler's object queue with the heap comprise the program's state.

The described language is mostly a deep embedding in Haskell: a continuation is the closed datatype `Stmt` whose abstract syntax is *interpreted* by the `eval` function. A pure continuation approach would be open to any IO statements as the continuation while probably yielding better performance. However, this deep embedding allows us to have multiple interpretations of the syntax: debug the syntax tree and have an equivalence result. Since we are focusing on concurrency, we deliberately avoid dealing with typing our target language; indeed, a GADT `Stm` syntax would guard our embedding from type mismatches between object and future references. An important thing to mention is that the `eval` function operates in "lockstep", i.e. one statement at a time. Modifying the function to execute from release point to release point (`await`, `get` and `return` from asynchronous calls) would benefit in performance but would affect the completeness w.r.t. the ABS semantics: some interleavings would be excluded. Finally, in this setting (and in our extract of the implementation) the global scheduler is sequentially simulated by a N:1 threading model of concurrency. Haskell's GHC runtime supports an M:N threading model allowing SMP parallelism, a feature exploited in the full ABS-to-Haskell compiler that we will use in the experiments.

4 Correctness

To prove that the translation is correct and resource preserving, we use an intermediate semantics $\rightarrowtail$ closer to the Haskell programs. This semantics, depicted in Fig. 7, considers configurations $(h, [\overline{o_m}])$ where all the information of the objects is stored in a unified heap—concretely $h(o_n)(\mathcal{Q})$ returns the process queue of object o_n. The semantics in Fig. 7 presents two main differences w.r.t. that in Fig. 2 and 3 of Sec. 2. First, the list $[\overline{o_m}]$ is used to apply a *round-robin* policy: the first unblocked object[3] o_n in $[\overline{o_m}]$ is selected using $nextObject(h, [\overline{o_m}])$, the first statement of the active process of o_n is executed and then the list is updated to continue with the object o_{n+1}. The other difference is that process queues do not contain sequences of statements but *continuations*, as explained in the previous section. To generate these continuation rules (Async) and (Sync) invoke the translation of the methods `m` with the adequate parameters. Nevertheless, the rules of the $\rightarrowtail$ semantics correspond with the semantic rules in Sec. 2.

Given a list $[\overline{o_m}]$ we use the notation $[\overline{o_{i \rightarrow k}}]$ for the sublist $[o_i, o_{i+i}, \ldots, o_k]$, and the operator (:) for list concatenation. In the rules (Async) and (Return$_A$),

[3] Object whose active process is not waiting for a future variable in a `get` statement.

$$\text{(Assign I)}\ \frac{\begin{array}{c} \mathit{nextObject}(h, [\overline{o_m}]) = o_n \quad h(o_n)(\mathcal{Q}) = (\texttt{Assign}\ x\ (\texttt{Attr}\ y)\ k', l) \cdot q \\ h' = h[(o_n)(x) \mapsto h(o_n)(y), (o_n)(\mathcal{Q}) \mapsto (k', l) \cdot q] \end{array}}{(h, [\overline{o_m}]) \rightarrowtail (h', [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}])}$$

$$\text{(Assign II)}\ \frac{\begin{array}{c} \mathit{nextObject}(h, [\overline{o_m}]) = o_n \quad h(o_n)(\mathcal{Q}) = (\texttt{Assign}\ x\ (\texttt{Param}\ r)\ k', l) \cdot q \\ h' = h[(o_n)(x) \mapsto r, (o_n)(\mathcal{Q}) \mapsto (k', l) \cdot q] \end{array}}{(h, [\overline{o_m}]) \rightarrowtail (h', [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}])}$$

$$\text{(New)}\ \frac{\begin{array}{c} \mathit{nextObject}(h, [\overline{o_m}]) = o_n \quad h(o_n)(\mathcal{Q}) = (\texttt{Assign}\ x\ \texttt{New}\ k', l) \cdot q \\ h(\mathit{count}) = o_{new} \quad h' = h[(o_n)(x) \mapsto o_{new}, \mathit{count} \mapsto o_{new} + 1, \\ (o_{new})(\mathcal{Q}) \mapsto \epsilon, (o_n)(\mathcal{Q}) \mapsto (k', l) \cdot q] \end{array}}{(h, [\overline{o_m}]) \rightarrowtail (h', [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}])}$$

$$\text{(Get)}\ \frac{\begin{array}{c} \mathit{nextObject}(h, [\overline{o_m}]) = o_n \quad h(o_n)(\mathcal{Q}) = (\texttt{Assign}\ x\ (\texttt{Get}\ f)\ k', l) \cdot q \\ h(h(o_n)(f)) = \texttt{Right}\ v \quad h' = h[(o_n)(x) \mapsto v, (o_n)(\mathcal{Q}) \mapsto (k', l) \cdot q] \end{array}}{(h, [\overline{o_m}]) \rightarrowtail (h', [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}])}$$

$$\text{(Await I)}\ \frac{\begin{array}{c} \mathit{nextObject}(h, [\overline{o_m}]) = o_n \quad h(o_n)(\mathcal{Q}) = (\texttt{Await}\ f\ k', l) \cdot q \\ h(h(o_n)(f)) = \texttt{Right}\ v \quad h' = h[(o_n)(\mathcal{Q}) \mapsto (k', l) \cdot q] \end{array}}{(h, [\overline{o_m}]) \rightarrowtail (h', [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}])}$$

$$\text{(Await II)}\ \frac{\begin{array}{c} \mathit{nextObject}(h, [\overline{o_m}]) = o_n \quad h(o_n)(\mathcal{Q}) = (\texttt{Await}\ f\ k', l) \cdot q \\ h(h(o_n)(f)) = \texttt{Left}\ e \quad h' = h[(o_n)(\mathcal{Q}) \mapsto q \cdot (k, l)] \end{array}}{(h, [\overline{o_m}]) \rightarrowtail (h', [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}])}$$

$$\text{(Async)}\ \frac{\begin{array}{c} \mathit{nextObject}(h, [\overline{o_m}]) = o_n \quad h(o_n)(\mathcal{Q}) = (\texttt{Assign}\ x\ (\texttt{Async}\ y\ m\ \bar{z})\ k', l) \cdot q \\ h(\mathit{count}) = l' \quad h(o_n)(y) = o_y \quad h(o_y)(\mathcal{Q}) = q_y \quad (m(\bar{w}) \mapsto S) \in D \\ k'' = \texttt{m}\ h(o_n)(\bar{z})\ o_n\ \texttt{Nothing undefined} \quad \mathit{newQ}_{add}([\overline{o_m}], o_n, o_y) = s \\ h' = h[(o_n)(x) \mapsto l', \mathit{count} \mapsto l' + 1, l' \mapsto \texttt{Left}\ [\,], \\ (o_n)(\mathcal{Q}) \mapsto (k', l) \cdot q, (o_y)(\mathcal{Q}) \mapsto q_y \cdot (k'', l')] \end{array}}{(h, [\overline{o_m}]) \rightarrowtail (h', s)}$$

$$\text{(Sync)}\ \frac{\begin{array}{c} \mathit{nextObject}(h, [\overline{o_m}]) = o_n \quad h(o_n)(\mathcal{Q}) = (\texttt{Assign}\ x\ (\texttt{Sync}\ m\ \bar{z})\ k', l) \cdot q \\ (m(\bar{w}) \mapsto S) \in D \quad k'' = \texttt{m}\ h(o_n)(\bar{z})\ o_n\ (\texttt{Just}\ x)\ k' \quad h' = h[(o_n)(\mathcal{Q}) \mapsto (k'', l) \cdot q] \end{array}}{(h, [\overline{o_m}]) \rightarrowtail (h', [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}])}$$

$$\text{(Return}_A\text{)}\ \frac{\begin{array}{c} \mathit{nextObject}(h, [\overline{o_m}]) = o_n \quad h(o_n)(\mathcal{Q}) = (\texttt{Return}\ z\ \texttt{Nothing}\ \_, l) \cdot q \\ \mathit{newQ}_{del}([\overline{o_m}], o_n, q) = s \quad h' = h[l \mapsto \texttt{Right}\ h(o_n)(z), (o_n)(\mathcal{Q}) \mapsto q] \end{array}}{(h, [\overline{o_m}]) \rightarrowtail (h', s)}$$

$$\text{(Return}_S\text{)}\ \frac{\begin{array}{c} \mathit{nextObject}(h, [\overline{o_m}]) = o_n \quad h(o_n)(\mathcal{Q}) = (\texttt{Return}\ z\ (\texttt{Just x})\ k', l) \cdot q \\ h' = h[(o_n)(x) \mapsto h(o_n)(z), (o_n)(\mathcal{Q}) \mapsto (k', l) \cdot q] \end{array}}{(h, [\overline{o_m}]) \rightarrowtail (h', [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}])}$$

Fig. 7: Intermediate semantics.

where the object list can increase or decrease one object, we use the following auxiliary functions. $\mathit{newQ}_{add}([\overline{o_m}], o_n, o_y)$ inserts the object o_y into $[\overline{o_m}]$ if it is new (i.e., it does not appear in $[\overline{o_m}]$), and $\mathit{newQ}_{del}([\overline{o_m}], o_n, q_n)$ removes the object o_n from $[\overline{o_m}]$ if its process queue q_n is empty. In both cases they advance the list of objects to o_{n+1}.

$$\mathit{newQ}_{add}([\overline{o_m}], o_n, o_y) = \begin{cases} [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}] & \text{if } o_y \in [\overline{o_m}] \\ [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}] : [o_y] & \text{if } o_y \notin [\overline{o_m}] \end{cases}$$

$$
\begin{array}{ll}
{}^{c}[\![\langle C, h\rangle]\!] = & (h', act), \text{where} \\
& act = [o_n \mid (o_n, Q_n) \in C, Q_n \neq \epsilon] \\
& C = \{(n_1, Q_1), \ldots, (n_m, Q_m)\} \text{ and} \\
& h' = h[\overline{(n_i)(\mathcal{Q}) \mapsto {}^{q}[\![Q_i]\!]}]
\end{array}
\qquad
\begin{array}{ll}
{}^{q}[\![\epsilon]\!] & = \epsilon \\
{}^{q}[\![(S, l) \cdot Q]\!] & = ({}^{s}[\![S]\!], l) \cdot {}^{q}[\![Q]\!]
\end{array}
$$

Fig. 8: Translation from source to target configurations.

$$
newQ_{del}([\overline{o_m}], o_n, q_n) = \begin{cases} [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n-1}}] & \text{if } q_n = \epsilon \\ [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}] & \text{if } q_n \neq \epsilon \end{cases}
$$

In order to reason about the different semantics, we define the translation from runtime configurations $\langle C, h\rangle$ of Sec. 2 to concrete Haskell data structures used in the intermediate $\rightarrowtail$ semantics and in the compiled Haskell programs (see Fig. 8). The set of closure lists C is translated into a list of object references, and the process queues inside C are included into the heap related to the special term $\mathcal{Q}$. Although we use the same notation h, we consider that the heap is translated into the corresponding Haskell tuple (*object_vector*, *future_vector*, *counter*) explained in Sec. 3. As usual with heaps, we use the notation $h[(o_n)(\mathcal{Q}) \mapsto q]$ to update the process queue of the object o_n to q. Finally, natural numbers become integers, global variables become Strings and $Nat_{\perp}$ values in the futures become *Either* values. To denote the inverse translation from data structures to runtime configurations we use ${}^{c}[\![(h', act)]\!]^{-1} = \langle C, h\rangle$—the same for queues ${}^{q}[\![\cdot]\!]^{-1}$ and statements ${}^{s}[\![\cdot]\!]^{-1}$. Note that the translation ${}^{c}[\![\cdot]\!]_{c}$ is not deterministic because it generates a list of object references from a set of closures C, so the order of the objects in the list is not defined. On the other hand, the translation of the heap in ${}^{c}[\![\cdot]\!]$ and the inverse translation ${}^{c}[\![\cdot]\!]^{-1}$ are deterministic.

Based on the previous definitions we can state the soundness of the traces, i.e., every trace of `eval` steps is a valid trace w.r.t. $\rightarrow$. Note that for the sake of conciseness we unify the statements S and their representation as Haskell terms `res`, since there is a straightforward translation between them. We consider the auxiliary function $updL([\overline{o_m}], o_n, l) = [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n-1}}] : l$ to update the list of object references.

Theorem 1 (Trace soundness). *Let (h_1, s_1) be an initial state and consider a sequence of $n-1$ consecutive* `eval` *steps defined as: a) $o_i = nextObject(h_i, s_i)$, b)* `(res`$_i$`, l`$_i$`, h`$_{i+1}$`) = eval o`$_i$ `h`$_i$*, c) $s_{i+1} = updL(s_i, o_i, l_i)$. Then ${}^{c}[\![(h_1, s_1)]\!]^{-1} \rightarrow^{o_1}_{res_1}$ ${}^{c}[\![(h_1, s_2)]\!]^{-1}_{c} \rightarrow^{o_2}_{res_2} \ldots \rightarrow^{o_{n-1}}_{res_{n-1}} {}^{c}[\![(h_n, s_n)]\!]^{-1}$.*

Note that it is not possible to obtain a similar result about trace completeness since the $\rightarrow$-semantics in Fig. 3 selects the next object to execute nondeterministic (random scheduler), whereas the intermediate $\rightarrowtail$-semantics in Fig. 7 follows a concrete *round-robin* scheduling policy. As a final remark notice that the intermediate semantics $\rightarrowtail$ can be seen as a *specification* of the `eval` function. Therefore it can be used to guide the correctness proof of `eval` using proof assistance tools like *Isabelle* [11] or to generate tests automatically using *QuickCheck* [6].

4.1 Preservation of Resource Consumption

A strong feature of our translation is that the Haskell-translated program preserves the *resource consumption* of the original ABS program. As in [1] we use the notion of *cost model* to parametrize the type of resource we want to bound. Cost models are functions from ABS statements to real numbers, i.e., $\mathcal{M} : S \to \mathbb{R}$ that define different resource consumption measures. For instance, if the resource to measure is the number of executed steps, $\mathcal{M} : S \to 1$ such that each instruction has cost one. However, if one wants to measure memory consumption, we have that $\mathcal{M}(new) = \texttt{obj\_size}$, where `obj_size` refers to the size of an object reference, and $\mathcal{M}(instr) = 0$ for all remaining instructions. The resource preservation is based on the notion of *trace cost*, i.e., the sum of the cost of the statements executed. Given a concrete cost model $\mathcal{M}$, an object reference o and a program execution $\mathcal{T} \equiv A_1 \to^{o_1}_{S_1} A_2 \to^{o_2}_{S_2} \ldots \to^{o_{n-1}}_{S_{n-1}} A_n$, the cost of the trace $\mathcal{C}(\mathcal{T}, o, \mathcal{M})$ is defined as

$$\mathcal{C}(\mathcal{T}, o, \mathcal{M}) = \sum_{S \in \mathcal{T}|_{\{o\}}} \mathcal{M}(S)$$

Notice that, from all the steps in the trace $\mathcal{T}$, it takes into account only those performed in object o (denoted as $\mathcal{T}|_{\{o\}}$), so the cost notion is *object-sensitive*. Since the trace soundness states that the `eval` function performs the same steps as some trace $\mathcal{T}$, the cost preservation is a straightforward corollary:

Corollary 1 (Consumption Preservation). *Let (h_1, s_1) be an initial state and consider a sequence $\mathcal{T}_E$ of $n-1$ consecutive* `eval` *steps defined as: a) $o_i = nextObject(h_i, s_i)$, b)* `(res_i, l_i, h_{i+1}) = eval o_i h_i`*, c) $s_{i+1} = updL(s_i, o_i, l_i)$. Then $\mathcal{T} \equiv {}^{c}[\![(h_1, s_1)]\!]^{-1} \to^{o_1}_{res_1} {}^{c}[\![(h_2, s_2)]\!]^{-1}_{c} \to^{o_2}_{res_2} \ldots \to^{o_{n-1}}_{res_{n-1}} {}^{c}[\![(h_n, s_n)]\!]^{-1}$ such that $\mathcal{C}(\mathcal{T}_E, o, \mathcal{M}) = \mathcal{C}(\mathcal{T}, o, \mathcal{M})$.*

As a side effect of the previous result, we know that the upper bounds that are inferred from the ABS programs (using resource analyzers like [1]) are valid upper bounds for the Haskell translated code. We denote by $UB_{main}()|_o$ the upper bound obtained for the analysis of a `main` method for the computation performed on object `o`.

Theorem 2 (Bound preservation). *Let P be a program, $\mathcal{T}_E$ a sequence of* `eval` *steps from an initial state (h_1, s_1) and $UB_{main}()|_o$ the upper bound obtained for the program P starting from the main block, restricted to the object o. Then $\mathcal{C}(\mathcal{T}_E, o, \mathcal{M}) \leq UB_{main}()|_o$*

5 Experimental Evaluation

In the previous section we proved that the execution of compiled Haskell programs has the same resource consumption as the original ABS traces, i.e., they execute the same statements in the same order and in the same objects. However, it is important that the compilation does not introduce an overhead during execution so that run-times are proportional to the steps executed. In order to

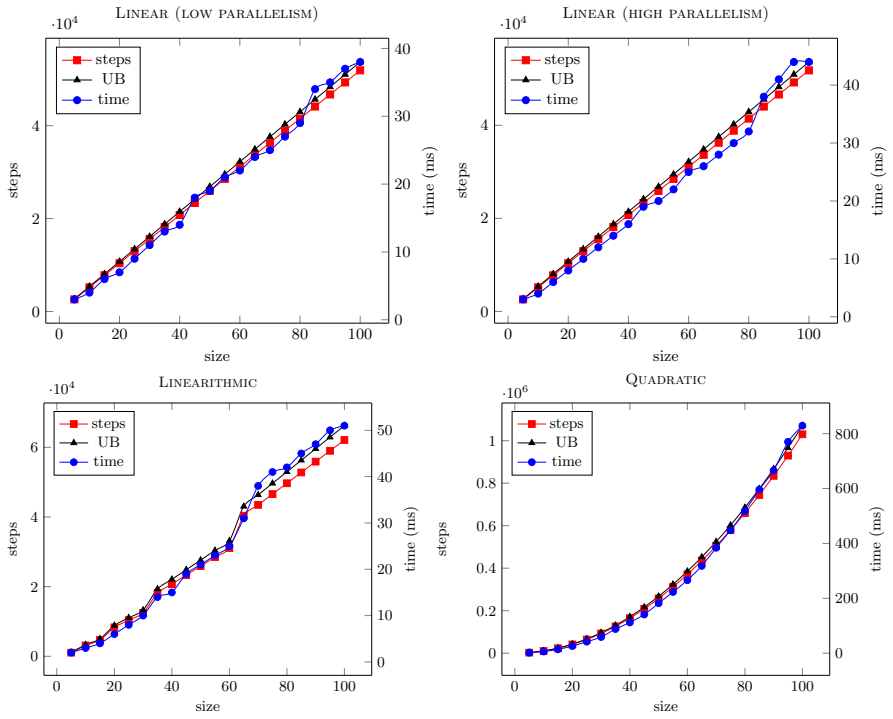

Fig. 9: Execution steps vs. time (Intel® Core™ i7-4790 at 3.60GHz, 16 GB).

evaluate this hypothesis, we have elaborated programs[4] with different asymptotic costs and measured the number of statements executed (steps) and their run-time. These experimental programs create a number n of objects (size) and invoke some tasks in each one: 1 task for the linear programs, *log* n tasks for the linearithmic program and n tasks for the quadratic program. The difference between the two linear programs is that the *low parallelism* version awaits for the result of the task before creating the next object, whereas the *high parallelism* version does not await. Fig. 9 shows the results of the tests as graphs, where the left vertical axis is used for the number of steps and the right vertical axis for the run-time in milliseconds. The graphs show that both the steps and time plots have the same growth rate in all the programs thus confirming the proportionality, i.e., the execution of one statement requires a constant amount of time. We have added to the graphs the resource bounds (UB) obtained by the SACO tool [2] from the analysis of the original ABS programs using the cost model that measures the number of statements executed. As can be appreciated, the bounds are higher than the actual number of steps but they are very precise for all the programs, except a small difference in the number of steps. This

[4] The ABS-subset experimental programs and measurements together with the proved Haskell embedding reside at https://github.com/abstools/abs-haskell-formal

	ABS Program	Haskell time	Haskell mem	Java time	Java mem	Erlang time	Erlang mem	Maude time	Maude mem
Sequential	BinarySearchTree	0.01	3716	0.3	73860	1.28	22824	0.5	47896
	NaiveFib	0.11	3924	18.38	476076	1.63	24796	198.62	38724
	Sequences	0.02	13308	8.76	787988	44.01	34036	2553.54	47948
	SumList	0.01	12312	1.06	190528	1.29	40716	1703.1	50832
Parallel	AwaitOnFut	0.05	6444	7.58	491952	1.91	26184	328.65	41908
	Bang	0.25	32956	5.91	734536	422.23	261776	396.49	40820
	BenchLists	5.97	18008	88.35	800404	58.67	371596	814.19	46984
	StressTest	0.04	6156	2.45	773312	8.30	75216	1213.04	45752
	SyncAsync	0.05	7312	24.14	1824676	13.97	377516	252.49	40144
	ThreadRingCOG	0.20	9036	37.9	1187388	284.71	166128	588.14	39324

Fig. 10: Benchmarks measuring time(s) and max memory(KB) for ABS backends targeting GHC7.10.1, OpenJDK1.8, Erlang18, Maude2.6 on Intel® i7-3537U, 8 GB

small imprecision in the upper bounds is caused by the constructors methods: the subset of the ABS language presented in this paper does not include constructors, but full ABS (and the SACO tool as well) considers that every object has a constructor. Therefore, the SACO tool will count a constant number of extra steps whenever a new object is created, corresponding to the invocation and execution of the implicit constructor.

The expectation from the Haskell backend presented in this paper was that its performance would surpass that of the other current ABS backends. For this, we compared the full-blown Haskell backend (completely covering the ABS language and using the "real" GHC multicore scheduler) against the Java, Erlang and Maude backends of ABS using a series of sequential and parallel ABS programs touching all features of the ABS language (can be found at https://github.com/abstools/abs-bench). The benchmark results shown in Fig. 10 indicate that the Haskell backend is indeed the fastest both in terms of elapsed time and memory residency. The Java backend is on average 166x slower while requiring 87x more memory than Haskell: the Java backend pays the price of Java's heavyweight threads coupled with spin-waiting when monitoring active objects for their *await* conditions. The Erlang backend takes 611x more time and 13x more memory, because the backend chose for a slower, process-oriented approach where each ABS process is implemented as a separate lightweight thread: the ABS processes of an active object are placed in a token ring—the process holding the token can execute unless it is blocked in which case the token is passed over causing needless spinning in certain cases. The Maude backend consumes comparable memory to Haskell but is extremely slow since the Maude interpreter is more suited for prototyping and model checking ABS semantics.

6 Conclusion

We have presented a concurrent object-oriented language (a subset of ABS) and its compilation to Haskell using continuations. The compilation is formalised in

order to show that the program behaviour and the resource consumption are preserved by the translation. We achieved this through a straightforward (one-to-one) mapping of source to target configurations, and statements to Haskell expressions. We show that this compilation does not introduce any overhead during execution, so that run-times are proportional to the number of steps executed, and that it outperforms previous compilers of ABS when considering the full set of the language. The translation is presented only for the core subset of the ABS language; it lacks features such as Algebraic Datatypes, "true" non-determinism, and multicore. Normally, to implement true non-determinism and multicore one needs system-level threads. This is indeed the case with our full-blown ABS-to-Haskell compiler (https://github.com/bezirg/abs2haskell).

In the future we plan to extend the correctness proof to the full-blown compiler, not only in terms of the omitted functional part of ABS, but regarding the behaviour of the non-deteterministic, multicore scheduler: this could be achieved by using either a high-level description of an "ideal" Haskell threaded scheduler or an abstraction of the actual lower-level GHC runtime scheduler. We nevertheless speculate that the resource-consumption outcomes will remain the same modulo the Garbage Collection. In a different direction, we would be interested to relate our resource preservation to the cloud extension for ABS [4]—a Cloud-Haskell [7] based extension to turn ABS to a distributed object system—and more specifically how the resource analysis results translate to network transport costs after any optimizations or network protocol limitations.

References

1. Albert, E., Arenas, P., Correas, J., Genaim, S., Gómez-Zamalloa, M., Puebla, G., Román-Díez, G.: Object-Sensitive Cost Analysis for Concurrent Objects. Softw. Test. Verif. Reliab. 25(3), 218–271 (2015)
2. Albert, E., Arenas, P., Flores-Montoya, A., Genaim, S., Gómez-Zamalloa, M., Martin-Martin, E., Puebla, G., Román-Díez, G.: SACO: Static Analyzer for Concurrent Objects. In: Proc. TACAS '14, pp. 562–567. LNCS 8413, Springer (2014)
3. Albert, E., Arenas, P., Gómez-Zamalloa, M.: Symbolic Execution of Concurrent Objects in CLP. In: Practical Aspects of Declarative Languages (PADL'12). pp. 123–137. LNCS 7149, Springer (2012)
4. Bezirgiannis, N., de Boer, F.S.: ABS: a high-level modeling language for Cloud-Aware Programming. In: Proc. SOFSEM '16. Springer (2016), to appear
5. de Boer, F.S., Clarke, D., Johnsen, E.B.: A Complete Guide to the Future. In: Proc. ESOP '07, pp. 316–330. LNCS 4421, Springer (2007)
6. Claessen, K., Hughes, J.: QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In: Proc. ICFP '00. pp. 268–279. ACM (2000)
7. Epstein, J., Black, A.P., Peyton-Jones, S.: Towards Haskell in the Cloud. In: ACM SIGPLAN Notices. vol. 46. ACM (2011)
8. Flanagan, C., Felleisen, M.: The Semantics of Future and its Use in Program Optimization. In: Proc. POPL '95. pp. 209–220. ACM (1995)
9. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A Core Language for Abstract Behavioral Specification. In: FMCO. pp. 142–164. LNCS 6957, Springer (2010)

10. Nakata, K., Saar, A.: Compiling Cooperative Task Management to Continuations. In: Proc. FSEN ' 13, pp. 95–110. LNCS 8161, Springer (2013)
11. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-order Logic. Springer-Verlag (2002)
12. Noll, T.: A Rewriting Logic Implementation of Erlang. ENTCS 44(2), 206–224 (2001)
13. Palacios, A., Vidal, G.: Towards Modelling Actor-Based Concurrency in Term Rewriting. In: Proc. WPTE '15. OASICS, vol. 46, pp. 19–29. Dagstuhl Pub. (2015)
14. Vidal, G.: Towards Erlang Verification by Term Rewriting. In: Proc. LOPSTR '13. pp. 109–126. LNCS 8901, Springer (2013)
15. Wong, P.Y., Albert, E., Muschevici, R., Proena, J., Schfer, J., Schlatte, R.: The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. STTT 14(5), 567–588 (2012)

***Note for the reviewers**: The following appendix contains the complete proofs of the theoretical results. It is not part of the paper. In case the paper is accepted, it will be made available as a technical report.*

A Proofs and auxiliary results

A.1 Auxiliary results

In this section we will state and prove the completeness and soundness of $\rightarrowtail$ w.r.t. $\rightarrow$. The completeness states that any $\rightarrow$-step can be performed in a translated Haskell term using $\rightarrowtail$ with the same object and statement. The soundness states that any $\rightarrowtail$-step is a valid $\rightarrow$-step from the translated configuration.

Lemma 1 (Completeness of $\rightarrowtail$). *If $A \rightarrow_S^{o_n} B$ then there are two Haskell tuples $t_A = {}^c[\![A]\!]$ and $t_B = {}^c[\![B]\!]$ such that $t_A \rightarrowtail_S^{o_n} t_B$.*

Proof. By case distinction on the rule used to perform the step.

- **(Internal)+(Assign I)**.

$$\text{(Internal)}\frac{\text{(Assign I)}\dfrac{h' = h[(o_n)(x) \mapsto h(o_n)(y)]}{\langle o_n : (\texttt{x:=y}; S, l) \cdot Q, h\rangle \rightarrow \langle o_n : (S, l) \cdot Q, h'\rangle}}{A \equiv \langle (o_n : (\texttt{x:=y}; S, l) \cdot Q) \cup C, h\rangle \rightarrow^{o_n}_{\texttt{x:=y}} \langle (o_n : (S, l) \cdot Q) \cup C, h'\rangle \equiv B}$$

  One possible translation of ${}^c[\![A]\!]$ would be $t_A = (h_c, [\overline{o_m}])$, where o_n is the first object in $\overline{o_m}$ that is not blocked and h_c is the heap h extended with the process queues $h_c = h[(o_m)(\mathcal{Q}) \mapsto {}^q[\![Q_m]\!]]$. Note that $h_c(o_n)(\mathcal{Q}) = {}^q[\![(\texttt{x:=y}; S, l) \cdot Q]\!] = (\texttt{Assign}\ x\ (\texttt{Attr}\ y)\ {}^s[\![S]\!], l) \cdot {}^q[\![Q]\!]$. Then from t_A we can perform a $\rightarrowtail$-step to t_B:

$$\text{(Assign I)}\frac{\begin{array}{c} nextObject(h_c, [\overline{o_m}]) = o_n \\ h_c(o_n)(\mathcal{Q}) = (\texttt{Assign}\ x\ (\texttt{Attr}\ y)\ {}^s[\![S]\!], l) \cdot {}^q[\![Q]\!] \\ h'_c = h_c[(o_n)(x) \mapsto h(o_n)(y), (o_n)(\mathcal{Q}) \mapsto ({}^s[\![S]\!], l) \cdot {}^q[\![Q]\!]] \end{array}}{t_A \equiv (h_c, [\overline{o_m}]) \rightarrowtail^{o_n}_{\texttt{x:=y}} (h'_c, [\overline{o_{n+1 \rightarrow m}}] : [\overline{o_{1 \rightarrow n}}]) \equiv t_B}$$

  Note that ${}^c[\![B]\!] = t_B$ since it contains the set of objects with references $\overline{o_m}$, which can be translated as the list $[\overline{o_{n+1 \rightarrow m}}] : [\overline{o_{1 \rightarrow n}}]$.
- **(Internal)+(Assign II)** and **(Internal)+(New)**. Similar to the previous case.
- **(Internal)+(Get)**.

$$\text{(Internal)}\frac{\text{(Get)}\dfrac{h(h(o_n)(f)) \neq \bot \qquad h' = h[(o_n)(x) \mapsto h(h(o_n)(f))]}{\langle o_n : (\texttt{x:=f.get}; S, l) \cdot Q, h\rangle \rightarrow \langle o_n : (S, l) \cdot Q, h'\rangle}}{\begin{array}{l} A \equiv \langle (o_n : (\texttt{x:=f.get}; S, l) \cdot Q) \cup C, h\rangle \rightarrow^{o_n}_{\texttt{x:=f.get}} \\ \langle (o_n : (S, l) \cdot Q) \cup C, h'\rangle \equiv B \end{array}}$$

  One possible translation of ${}^c[\![A]\!]$ would be $t_A = (h_c, [\overline{o_m}])$, where o_n is the first object in $\overline{o_m}$ that is not blocked and h_c is the heap h extended

with the process queues $h_c = h[\overline{(o_m)(\mathcal{Q}) \mapsto {}^q[\![Q_m]\!]}]$. Note that $h_c(o_n)(\mathcal{Q}) = {}^q[\![(\texttt{x:=f.get}; S, l) \cdot Q]\!] = (\texttt{Assign}\ x\ (\texttt{Get}\ f)\ {}^s[\![S]\!], l) \cdot {}^q[\![Q]\!]$. Then from t_A we can perform a $\rightarrowtail$-step to t_B:

$$(\textsc{Get})\ \frac{\begin{array}{c} nextObject(h_c, [\overline{o_m}]) = o_n \\ h_c(o_n)(\mathcal{Q}) = (\texttt{Assign}\ x\ (\texttt{Get}\ f)\ {}^s[\![S]\!], l) \cdot {}^q[\![Q]\!] \\ h_c(h_c(o_n)(f)) = \texttt{Just}\ v \\ h'_c = h_c[(o_n)(x) \mapsto v, (o_n)(\mathcal{Q}) \mapsto ({}^s[\![S]\!], l) \cdot {}^q[\![Q]\!]] \end{array}}{t_A \equiv (h_c, [\overline{o_m}]) \rightarrowtail^{o_n}_{\texttt{x:=f.get}} (h'_c, [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}]) \equiv t_B}$$

and ${}^c[\![B]\!] = t_B$.

- **(Internal)+(Await I)** and **(Internal)+(Await II)**. Similar to the previous case, with the main difference that (Await I) inserts the current process in the first position of the queue, as usual, and (Await II) at the end.
- **(Message)+(Async)**.

$$(\textsc{Message})\ \frac{\begin{array}{c} \langle o_n : (\texttt{f:=x!m}(\bar{z}); S, l) \cdot Q_n, h\rangle \xrightarrow{o_d.m(l', \bar{r})} \langle o_n : (S, l) \cdot Q_n, h'\rangle \\ m(\bar{w}) \mapsto S_m \in D \qquad \tau = [\bar{w} \mapsto \bar{r}] \qquad S' = \widehat{(S_m \tau)}^* \end{array}}{\begin{array}{l} A \equiv \langle (o_n : (\texttt{f:=x!m}(\bar{z}); S, l) \cdot Q_n) \cup (o_d : Q_d) \cup C, h\rangle \rightarrow^{o_n}_{\texttt{f:=x!m}(\bar{z})} \\ \langle (o_n : (S, l) \cdot Q_n) \cup (o_d : Q_d \cdot (S', l')) \cup C, h'\rangle \equiv B \end{array}}$$

where

$$(\textsc{Async})\ \frac{\begin{array}{c} h(o_n)(x) = d \qquad h(count) = l' \qquad \bar{r} = h(o_n)(\bar{z}) \\ h' = h[(o_n)(f) \mapsto l', (l') \mapsto \bot, count \mapsto l' + 1] \end{array}}{\langle o_n : (\texttt{f:=x!m}(\bar{z}); S, l) \cdot Q_n, h\rangle \xrightarrow{o_d.m(l', \bar{r})} \langle o_n : (S, l) \cdot Q_n, h'\rangle}$$

One possible translation of ${}^c[\![A]\!]$ is $t_A = (h_c, [\overline{o_m}])$, where o_n is the first object in $\overline{o_m}$ that is not blocked and h_c is the heap h extended with the process queues $h_c = h[\overline{(o_m)(\mathcal{Q}) \mapsto {}^q[\![Q_m]\!]}]$. Note that:

- $h_c(o_n)(\mathcal{Q}) = (\texttt{Assign}\ x\ (\texttt{Async}\ x\ m\ \bar{z})\ {}^s[\![S]\!], l) \cdot {}^q[\![Q_n]\!]$
- $h_c(o_d)(\mathcal{Q}) = {}^q[\![Q_d]\!]$

Then from ${}^c[\![A]\!]$ we can perform a $\rightarrowtail$-step to t_B:

$$(\textsc{Async})\ \frac{\begin{array}{c} nextObject(h_c, [\overline{o_m}]) = o_n \qquad h(count) = l' \\ h_c(o_n)(\mathcal{Q}) = (\texttt{Assign}\ f\ (\texttt{Async}\ x\ m\ \bar{z})\ {}^s[\![S]\!], l) \cdot {}^q[\![Q_n]\!] \\ h_c(o_n)(x) = d \qquad h_c(d)(\mathcal{Q}) = {}^q[\![Q_d]\!] \qquad (m(\bar{w}) \mapsto S_m) \in D \\ k = \texttt{m}\ h_c(o_n)(\bar{z})\ o_n\ \texttt{Nothing undefined} \\ newQ_{add}([\overline{o_m}], o_n, d) = s \\ h'_c = h_c[(o_n)(f) \mapsto l', count \mapsto l' + 1, (l') \mapsto \bot, \\ (o_n)(\mathcal{Q}) \mapsto ({}^s[\![S]\!], l) \cdot {}^q[\![Q_n]\!], (d)(\mathcal{Q}) \mapsto {}^q[\![Q_d]\!] \cdot (k, l')] \end{array}}{t_A \equiv (h_c, [\overline{o_m}]) \rightarrowtail^{o_n}_{\texttt{f:=x!m}(\bar{z})} (h'_c, s) \equiv t_B}$$

where ${}^c[\![B]\!] = t_B$. Note that by the definition of ${}^m[\![\cdot]\!]$ and ${}^s[\![\cdot]\!]$

$$k = \texttt{m}\ h_c(o_n)(\bar{z})\ o_n\ \texttt{Nothing undefined}) = {}^s[\![S']\!] = {}^s[\![S']\!]_{\texttt{undefined,Nothing}}$$

so ${}^q[\![Q_d \cdot (S', l')]\!] = {}^q[\![Q_d]\!] \cdot ({}^s[\![S']\!], l') = {}^q[\![Q_d]\!] \cdot (k, l')$. On the other hand, by construction s is a list of those object references whose queues ($\mathcal{Q}$) are not empty.

– **(Internal)+(Sync)**.

$$\text{(Internal)}\frac{\text{(Sync)}\ \dfrac{(m(\bar{w}) \mapsto S_m) \in D \ \textit{fresh} \qquad \tau = [\bar{w} \mapsto h(n)(\bar{z})] \qquad S' = \widehat{(S_m\tau)}^{\,x}}{\langle o_n : (\texttt{x:=m}(\bar{z}); S, l) \cdot Q, h\rangle \to \langle o_n : (S'; S, l) \cdot Q, h\rangle}}{\begin{array}{l} A \equiv \langle (o_n : (\texttt{x:=m}(\bar{z}); S, l) \cdot Q) \cup C, h\rangle \to^{o_n}_{\texttt{x:=m}(\bar{z})} \\ \langle (o_n : (S'; S, l) \cdot Q) \cup C, h\rangle \equiv B \end{array}}$$

One possible translation of ${}^{c}[\![A]\!]$ is $t_A = (h_c, [\overline{o_m}])$, where o_n is the first object in $\overline{o_m}$ that is not blocked and h_c is the heap h extended with the process queues $h_c = h[\overline{(o_m)(\mathcal{Q}) \mapsto {}^{q}[\![Q_m]\!]}]$. Note that $h_c(o_n)(\mathcal{Q}) = (\texttt{Assign}\ x\ (\texttt{Sync}\ m\ \bar{z})\ {}^{s}[\![S]\!], l) \cdot {}^{q}[\![Q]\!]$. Then from t_A we can perform a $\rightarrowtail$-step to t_B:

$$\text{(Sync)}\frac{\begin{array}{c} \textit{nextObject}(h, [\overline{o_m}]) = o_n \\ h_c(o_n)(\mathcal{Q}) = (\texttt{Assign}\ x\ (\texttt{Sync}\ m\ \bar{z})\ {}^{s}[\![S]\!], l) \cdot {}^{q}[\![Q]\!] \\ k = \texttt{m}(h(o_n)(\bar{z}), o_n, \texttt{Just}\ x, {}^{s}[\![S]\!]) \\ h' = h[(o_n)(\mathcal{Q}) \mapsto (k, l) : {}^{q}[\![Q]\!]] \end{array}}{t_A \equiv (h, [\overline{o_m}]) \rightarrowtail (h', [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}]) \equiv t_B}$$

where ${}^{c}[\![B]\!] = t_B$. Note that by definition of ${}^{m}[\![\cdot]\!]$ and the translation ${}^{s}[\![\cdot]\!]$

$$k = \texttt{m}(h_c(o_n)(\bar{z}), o_n, \texttt{Just}\ x, {}^{s}[\![S]\!]) = {}^{s}[\![\widehat{S_m\tau}^{\,x}]\!]_{({}^{s}[\![S]\!])}$$

so $k = {}^{s}[\![\widehat{S_m\tau}^{\,x}; S]\!]$.

– **(Internal)+(Return$_A$)**.

$$\text{(Internal)}\frac{\text{(Return}_A\text{)}\ \dfrac{h' = h[(l) \mapsto h(o_n)(x)]}{\langle o_n : (\texttt{return x}; S, l) \cdot Q, h\rangle \to \langle o_n : Q, h'\rangle}}{\begin{array}{l} A \equiv \langle (o_n : (\texttt{return x}; S, l) \cdot Q) \cup C, h\rangle \to^{o_n}_{\texttt{return x}} \\ \langle (o_n : (S'; S, l) \cdot Q) \cup C, h\rangle \equiv B \end{array}}$$

One possible translation of ${}^{c}[\![A]\!]$ is $t_A = (h_c, [\overline{o_m}])$, where o_n is the first object in $\overline{o_m}$ that is not blocked and h_c is the heap h extended with the process queues $h_c = h[\overline{(o_m)(\mathcal{Q}) \mapsto {}^{q}[\![Q_m]\!]}]$. Note that $h_c(o_n)(\mathcal{Q}) = (\texttt{Return}\ x\ \texttt{Nothing}\ {}^{s}[\![S]\!], l) \cdot {}^{q}[\![Q]\!]$. Then from t_A we can perform a $\rightarrowtail$-step to t_B:

$$\text{(Return}_A\text{)}\frac{\begin{array}{c} \textit{nextObject}(h_c, [\overline{o_m}]) = o_n \\ h_c(o_n)(\mathcal{Q}) = (\texttt{Return}\ x\ \texttt{Nothing}\ {}^{s}[\![S]\!], l) \cdot {}^{q}[\![Q]\!] \\ \textit{newQ}_{del}([\overline{o_m}], o_n, {}^{q}[\![Q]\!]) = s \\ h'_c = h_c[l \mapsto h(o_n)(x), (o_n)(\mathcal{Q}) \mapsto {}^{q}[\![Q]\!]] \end{array}}{t_A \equiv (h_c, [\overline{o_m}]) \rightarrowtail (h'_c, s) \equiv t_B}$$

where ${}^{c}[\![B]\!] = t_B$. Note that s will not contain o_n if ${}^{q}[\![Q]\!]$ is empty.

– **(Internal)+(Return$_S$)**. Similar to the previous case.

□

Lemma 2 (Soundness of $\rightarrowtail$). *If* $t_A \rightarrowtail^{o_n}_{S} t_B$ *then* ${}^{c}[\![t_A]\!]^{-1} \to^{o_n}_{S} {}^{c}[\![t_B]\!]^{-1}$.

Proof. By case distinction on the rule applied to perform the step. The reasoning is very similar to the proof of Theorem 1 so we only include the case of (Assing I); the other rules follow the same ideas.

– **(Assign I)**.

$$\text{(Assign I)}\ \frac{\begin{array}{c} nextObject(h_c, [\overline{o_m}]) = o_n \qquad h_c(o_n)(\mathcal{Q}) = (\texttt{Assign}\ x\ (\texttt{Attr}\ y)\ k', l) : q \\ h'_c = h_c[(o_n)(x) \mapsto h(o_n)(y), (o_n)(\mathcal{Q}) \mapsto (k', l) : q] \end{array}}{t_A \equiv (h_c, [\overline{o_m}]) \rightarrowtail^{o_n}_{\texttt{x:=y}} (h'_c, [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}]) \equiv t_B}$$

The inverse translation of t_A is defined as

$$A = {}^{c}[\![t_A]\!]^{-1} = ((o_n : (\texttt{x:=y}; S, l) \cdot Q) \cup C, h_c)$$

where ${}^{s}[\![k']\!]^{-1} = S$, ${}^{q}[\![q]\!]^{-1} = Q$, h_c is the inverse translation of h and C is the inverse translation of the rest of object queues. Then from A we can perform the following derivation:

$$\text{(Internal)}\ \frac{\text{(Assign I)}\ \dfrac{h' = h[(o_n)(x) \mapsto h(o_n)(y)]}{\langle o_n : (\texttt{x:=y}; S, l) \cdot Q, h\rangle \rightarrow \langle o_n : (S, l) \cdot Q, h'\rangle}}{A \equiv \langle (o_n : (\texttt{x:=y}; S, l) \cdot Q) \cup C, h\rangle \rightarrow^{o_n}_{\texttt{x:=y}} \langle (o_n : (S, l) \cdot Q) \cup C, h'\rangle \equiv B}$$

It is clear that ${}^{c}[\![t_B]\!]^{-1} = B$ as the set of object referencies in B is $\{\overline{o_m}\}$ and h'_c is the same as h_c with the following changes: a) $h'_c(o_n)(\mathcal{Q}) = {}^{q}[\![h_c(o_n)(\mathcal{Q})]\!]^{-1} = (S, l) \cdot Q$ and b) $h'_c(o_n)(x) = h_c(o_n)(y)$.

□

Similar results can be stated about the compiled Haskell programs w.r.t. $\rightarrowtail$. The completeness states that any $\rightarrowtail$-step is performed by the `eval` function in the compiled program, and the soundness states that the result of `eval` is a valid $\rightarrowtail$-step when applied to the next unblocked object returned by the *nextObject* function.

Lemma 3 (Completeness of the compilation). *If* $(h, [\overline{o_m}]) \rightarrowtail^{o_n}_{res} (h', [\overline{o_k}])$ *then* `eval` o_n `h = (res,l,h')` *such that* $[\overline{o_k}] \equiv updL([\overline{o_m}], o_n, l)$.

Proof. The `eval` function is defined in file *Eval.hs* in the repository https://github.com/abstools/abs-haskell-formal/blob/master/src/Eval.hs. The first lines of the `eval` function extracts the information `(attrs,pqueue)` of object `this` from the heap, selects the first process from `pqueue` and selects its first continuation `c`. Note that the datatype `Data.Sequence` is imported with name `S`, and that we assume that `this` $= o_n$, i.e., the object at position n in $[\overline{o_m}]$.

```
1 eval this h = do
2   (attrs,pqueue) <- objects h `V.read` this
3   case S.viewl pqueue of
4     S.EmptyL -> error "(...)"
5     (Proc (destiny, c)) S.:< restProcs -> let res = c
6                                           in case res of
```

Then we proceed by case distinction on the rule used to perform the $\rightarrowtail$-step.

– **(Assign I)**.

$$(\textsc{Assign I})\frac{\begin{array}{c} nextObject(h, [\overline{o_m}]) = o_n \qquad h(o_n)(\mathcal{Q}) = (k, l) : q \\ k = \texttt{Assign}\ x\ (\texttt{Attr}\ y)\ k' \\ h' = h[(o_n)(x) \mapsto h(o_n)(y), (o_n)(\mathcal{Q}) \mapsto (k', l) : q] \end{array}}{(h, [\overline{o_m}]) \rightarrowtail (h', [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}])}$$

If $k = \texttt{Assign}\ x\ (\texttt{Attr}\ y)\ k'$ then `res` will be `Assign lhs (Attr a) k'` where `lhs` and `a` are the position in the vector `attrs` of the variables `x` and `y` respectively, therefore the **case** `ref` **of** expression will execute the following branch:

```
Assign lhs (Attr a) k' -> do
  (attrs `V.write` lhs) =<< (attrs `V.read` a)
  updateObj $ Left k'
  return (res,
          [this],
          h)
```

The heap is updated to store the value of `y` in `x` using the vector operators `V.read` and `V.write`, and the process is updated to have the continuation `k'` in the front—see definition of the `updateObj` function. Finally it returns the instruction `res`, the unitary list `[this]` and the new heap `h`—note that it has been updated, so $\texttt{h} = h'$. Clearly $[\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}] \equiv [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n-1}}] : l$ since $l \equiv [o_n]$.

– **(Assign II)**. Similar to the previous case, but using directly the value `r` inside the instruction `res` to update `lhs`.

– **(New)**.

$$(\textsc{New})\frac{\begin{array}{c} nextObject(h, [\overline{o_m}]) = o_n \qquad h(o_n)(\mathcal{Q}) = (k, l) : q \\ k = \texttt{Assign}\ x\ \texttt{New}\ k' \qquad h(count) = o_{new} \\ h' = h[(o_n)(x) \mapsto o_{new}, count \mapsto o_{new} + 1, \\ (o_{new})(\mathcal{Q}) \mapsto \epsilon, (o_n)(\mathcal{Q}) \mapsto (k', l) : q] \end{array}}{(h, [\overline{o_m}]) \rightarrowtail (h', [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}])}$$

If $k = \texttt{Assign}\ x\ \texttt{New}\ k'$ then `res` will be `Assign lhs New k'` where `lhs` is the position in the vector `attrs` of the variable `x`. The **case** `ref` **of** expression will follow the branch:

```
Assign lhs New k' -> do
  (attrs `V.write` lhs) $ newRef h
  updateObj $ Left k'
  initAttrVec <- V.replicate 10 (-1)
  (objects h `V.write` newRef h) (initAttrVec, S.empty)
  h' <- incCounterMaybeGrow
  return (res,
          [this],
          h')
```

This code updates the heap by storing a fresh reference (the function `newRef` extracts it from the heap) in the variable `x` (line 14), and, as in the assignment case, it updates the process queue pushing the next continuation `k'` in the front using function `updateObj` (line 15). In lines 16–17 the code creates an initial mapping `initAttrVec` for the new object and inserts in the heap with an empty process queue `S.empty`. Finally it increments the reference counter using the function `incCounterMaybeGrow`[5] and returns `(res,[this],h')`. It is clear that $h' = \texttt{h'}$ and $[\overline{o_{n+1\to m}}] : [\overline{o_{1\to n}}] \equiv [\overline{o_{n+1\to m}}] : [\overline{o_{1\to n-1}}] : l$ since $l \equiv [o_n]$.

– **(Get)**.

$$(\textsc{Get})\frac{\begin{array}{c} \mathit{nextObject}(h, [\overline{o_m}]) = o_n \qquad h(o_n)(\mathcal{Q}) = (k, l) : q \\ k = \texttt{Assign}\ x\ (\texttt{Get}\ f)\ k' \qquad h(h(o_n)(f)) = \texttt{Right}\ v \\ h' = h[(o_n)(x) \mapsto v, (o_n)(\mathcal{Q}) \mapsto (k', l) : q] \end{array}}{(h, [\overline{o_m}]) \rightarrowtail (h', [\overline{o_{n+1\to m}}] : [\overline{o_{1\to n}}])}$$

If $k = \texttt{Assign}\ x\ (\texttt{Get}\ y)\ k'$ then `res` will be `Assign lhs (Get a) k'` where `lhs` and `a` are the position in the vector `attrs` of the variables `x` and `y` respectively. In this case the **case `ref` of** expression will execute the following branch:

```
22 Assign lhs (Get a) k' -> do
23   f <- attrs `V.read` a
24   fval <- (futures h) `V.read` f
25   case fval of
26     -- unresolved future
27     Left blockedCallers -> do
28       (...)
29     -- already-resolved future
30     Right v -> do
31       (attrs `V.write` lhs) v
32       updateObj $ Left k'
33       return (res,
34               [this],
35               h)
```

The code fetchs the value `fval` of the future stored in the reference that appears in the variable `y` (lines 23–24). Since the future is resolved to a value due to the premises of the (Get) rule—`fval = Right v`—the value is stored in the variable `x` and the process queue is updated by pushing the next continuation `k'` in the front using function `updateObj` (lines 31–32). Finally, it returns `(res,[this],h)`. As in the previous cases it is straighforward to prove that the new heap `h`—which has been updated in place—is equal to h' and $[\overline{o_{n+1\to m}}] : [\overline{o_{1\to n}}] \equiv [\overline{o_{n+1\to m}}] : [\overline{o_{1\to n-1}}] : l$ since $l \equiv [o_n]$. The code ommited in line 28 handles when the future is not resolved, i.e., when `fval = Left blockedCallers`, situation that cannot happen considering the premises of the (Get) rule.

[5] Since the implementation uses *growable arrays* to store the mapping from objects to their attributes, this function also checks if the array is complete and must grow.

– **(Await I)**.

$$\text{(Await I)}\ \frac{\begin{array}{c} nextObject(h, [\overline{o_m}]) = o_n \quad h(o_n)(\mathcal{Q}) = (k, l) : q \\ k = \texttt{Await}\ f\ k' \quad h(h(o_n)(f)) = \texttt{Right}\ v \\ h' = h[(o_n)(\mathcal{Q}) \mapsto (k', l) : q] \end{array}}{(h, [\overline{o_m}]) \rightarrowtail (h', [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}])}$$

Then `res` is `Await attr k'`, where `attr` is the position in the vector `attrs` of the future variable `f`. The `eval` function will enter into the following branch:

```
36 Await attr k' -> do
37   fut <- V.read (futures h) =<< (attrs `V.read` attr)
38   case fut of
39     -- unresolved future
40     Left _ -> do
41       updateObj $ Right c
42       return (res,
43               [this],
44               h)
45     -- already-resolved future
46       Right _ -> do
47         updateObj $ Left k'
48         return (res,
49                 [this],
50                 h)
```

The variable `fut` contains the value stored in the future variable, which must be **Right _** because the rule (Await I) has been applied. The branch in lines 46–50 updates the heap `h` by storing the continuation `k'` in the front of the process queue and return `(res,[this],h)`. The updated heap `h` is equal to h', and clearly $[\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}] \equiv [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n-1}}] : o_n$.

– **(Await II)**. Similar to the (Await II) case, but `fut` must be **Left _** because the future is undefined. Then the branch in lines40–44 updates the heap `h` by storing the original continuation `c` in the back of the process queue—see function `updateObj` the the parameter is **Right c**.

– **(Async)**.

$$\text{(Async)}\ \frac{\begin{array}{c} nextObject(h, [\overline{o_m}]) = o_n \quad h(o_n)(\mathcal{Q}) = (k, l) : q \quad h(count) = l' \\ k = \texttt{Assign}\ x\ (\texttt{Async}\ y\ m\ \bar{z})\ k' \quad h(o_n)(y) = o_y \quad h(o_y)(\mathcal{Q}) = q_y \\ (m(\bar{w}) \mapsto S) \in D \quad k'' = \texttt{m}(h(o_n)(\bar{z}), o_n, \texttt{Nothing}, \lambda\ \emptyset \to \texttt{undefined}) \\ newQ_{add}([\overline{o_m}], o_n, o_y) = s \\ h' = h[(o_n)(x) \mapsto l', count \mapsto l' + 1, l' \mapsto \texttt{Left}\ [\,], \\ (o_n)(\mathcal{Q}) \mapsto (k', l) : q, (o_y)(\mathcal{Q}) \mapsto q_y : (k'', l')] \end{array}}{(h, [\overline{o_m}]) \rightarrowtail (h', s)}$$

Then `res` will have the value `Assign lhs (Async obj m params) k'`, where:

- `lhs` and `obj` are the positions of x and y in the vector `attrs`
- `m` is the Haskell function that is the translation of method m

- `params` is a list of variables (the arguments of the method invocation)
- `k'` is the continuation

The execution of `eval` will follow this branch:

```
51 Assign lhs (Async obj m params) k' -> do
52   calleeObj <- attrs 'V.read' obj -- read the callee object
53   ( calleeAttrs , calleeProcQueue) <- (objects h 'V.read' calleeObj)
54   derefed_params <- mapM (attrs 'V.read') params -- read the passed attrs
55   let newCont = m
56                 derefed_params
57                 calleeObj
58                 Nothing -- no writeback
59                 (error "...")
60   ( attrs 'V.write' lhs) (newRef h)
61   updateObj (Left k')
62   let newProc = Proc (newRef h, newCont)
63   ( objects h 'V.write' calleeObj) ( calleeAttrs , calleeProcQueue S.|> newProc)
64   ( futures h 'V.write' newRef h) (Left [ ])  -- create a new unresolved future
65   h' <- incCounterMaybeGrow
66   return (res,
67           this :[ calleeObj   | S.null calleeProcQueue],
68           h')
```

The first 3 lines obtain the mapping and process queue of object `obj` and create a list of reference values from the list of variables (`derefed_params`). Lines 55–59 invokes `m` to obtain the continuation `newCont` related to the asynchronous call. Line 60 stores the new reference `newRef h` in the variable `lhs`, and line 61 updates the heap by inserting the continuation `k'` in the front of the process queue of the current object. The next two lines creates and inserts in the back of the process queue of object `obj` a new process with continuation `newCont` and destiny the new reference `newRef h`. Line 64 creates a new undefined future variable, i.e., with value **Left** `[ ]`, and line 65 increments the reference counter of the heap—recall that as mappings are implemented as *growable arrays* the function `incCounterMaybeGrow` can increment their size. Finally, a tuple with the instruction `res`, a list of objects and the new heap `h'` is returned.

It is easy to see that `h'` is equal to h' since they have received the same updates. If $o_y \in [\overline{o_m}]$ then $s = [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}]$. In this case `calleeProcQueue` must not be empty, so the list of objects returned will be `[this]` and $s = [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n-1}}] : o_n$—recall that `this`$=o_n$. On the other hand if $o_y \notin [\overline{o_m}]$ then $s = [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}] : o_y$, so `calleeProcQueue` must be empty and the list of objects returned will be `[this,obj]`. Therefore $s = [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n-1}}] : [o_n, o_y]$—recall that $o_y =$`obj`.

– **(Sync)**.

$$\text{(Sync)}\ \frac{\begin{array}{c} \mathit{nextObject}(h, [\overline{o_m}]) = o_n \qquad h(o_n)(\mathcal{Q}) = (k, l) : q \\ k = \texttt{Assign}\ x\ (\texttt{Sync}\ m\ \bar{z})\ k' \qquad (m(\bar{w}) \mapsto S) \in D \\ k'' = \texttt{m}(h(o_n)(\bar{z}), o_n, \texttt{Just}\ x, k') \\ h' = h[(o_n)(\mathcal{Q}) \mapsto (k'', l) : q] \end{array}}{(h, [\overline{o_m}]) \rightarrowtail (h', [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}])}$$

In this case `res` will be `Assign lhs (Sync m params) k'` and the execution of `eval` will follow the branch:

```
69 Assign lhs (Sync m params) k' -> do
70   derefed_params <- mapM (attrs `V.read`) params -- read the passed attrs
71   updateObj $ Left (m
72                     derefed_params
73                     this
74                     (Just lhs)
75                     k')
76   return (res,
77           [this],
78           h)
```

The resoning is similar to the (ASYNC) case, but the new continuation related to the invocation is inserted in the front of the process queue of the current object—function `updateObj` in line 71.

– **(Return$_A$)**.

$$(\textsc{Return}_A)\frac{\begin{array}{c} nextObject(h, [\overline{o_m}]) = o_n \qquad h(o_n)(\mathcal{Q}) = (k, l) : q \\ k = \texttt{Return}\ z\ \texttt{Nothing}\ \_ \qquad newQ_{del}([\overline{o_m}], o_n, q) = s \\ h' = h[l \mapsto \texttt{Right}\ h(o_n)(z), (o_n)(\mathcal{Q}) \mapsto q] \end{array}}{(h, [\overline{o_m}]) \rightarrowtail (h', s)}$$

In this case `res = Return attr wb k'`, where `attr` is the position of the variable z in the mapping, `wb` is the *write-back* variable (or **Nothing** in asynchronous calls) and `k'` is the continuation to execute in the current process after returning. The execution of `eval` will follow the branch:

```
79 Return attr wb k' -> case wb of
80   -- sync call
81   Just lhs -> do
82     (attrs `V.write` lhs) =<< (attrs `V.read` attr)
83     updateObj $ Left k'
84     return (res,
85             [this],
86             h
87             )
88   -- async call
89   Nothing -> do
90     fut <- futures h `V.read` destiny
91     case fut of
92       Right _ -> error "..."
93       Left blockedCallers -> do
94         (futures h `V.write` destiny) =<< liftM Right (attrs `V.read` attr)
95         (objects h `V.write` this) (attrs, restProcs)
96         return (res,
97                 [this | not $ S.null restProcs] ++ blockedCallers,
98                 h)
```

Since the rule (RETURN$_A$) has been applied, then `wb = Nothing` and the inner branch in lines 89-98 is executed. Following defensive programming techniques, the code first checks that the future variable where the value is stored does not contain any previous value, i.e, it stores **Left** `e`, and throws an error otherwise. However, it is guaranteed that in any sequence of $\rightarrowtail$-steps the future variable will be unresolved when executing a **return** step: only one **return** will be executed in a process and future variables are not reused. Therefore the branch in lines 93–98 will be executed. First, the value of z (position `attr`) is stored in the future variable in position `destiny`—recall that `destiny` is the position of the future variable l from the (RETURN$_A$) rule, see line 5. Then in line 95 it removes the current process from the process queue in the `this` object, and in lines 96–98 it return the result tuple. Note that `blockedCallers` is an empty list: it is created empty when creating an asynchronous call—see the case for the (ASYNC) rule—and it is not modified in other instruction. However the code includes `blockedCallers` because it has been prepared to incorporate some optimizations in the future for handling efficiently those objects blocked waiting for future variables in a `get` instruction. It is straightforward to check that the updated heap `h` is the same as the new heap h' from the (RETURN$_A$) rule, as both have received the same updates. By definition of $newQ_{del}$ if $q_n = \epsilon$ then $s = [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n-1}}]$. In this case $s = [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n-1}}] : [\,]$ because `restProcs` will be **null**. On the other hand, if $q_n \neq \epsilon$ then $s = [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}]$ and clearly $s = [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n-1}}] : [o_n]$ because `restProcs` will not be **null**.

– **(Return$_S$)**.

$$(\text{RETURN}_S)\frac{\begin{array}{c} \mathit{nextObject}(h, [\overline{o_m}]) = o_n \qquad h(o_n)(\mathcal{Q}) = (k, l) : q \\ k = \texttt{Return}\ z\ (\texttt{Just}\ \texttt{x})\ k' \\ h' = h[(o_n)(x) \mapsto h(o_n)(z), (o_n)(\mathcal{Q}) \mapsto (k', l) : q] \end{array}}{(h, [\overline{o_m}]) \rightarrowtail (h', [\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}])}$$

Similar to the previous case but executing the branch in lines 81–87: the returned value is stored in the `lhs` variable (line 82), and the current process continues with the new continuation `k'` (line 83), which is inserted in the front of the process queue. □

Lemma 4 (Soundness of the compilation). *If* `eval` o_n `h = (res,l,h')` *and* $\mathit{nextObject}(h, [\overline{o_m}]) = o_n$ *then* $(h, [\overline{o_m}]) \rightarrowtail^{o_n}_{res} (h', \mathit{updL}([\overline{o_m}], o_n, l))$.

Proof. By case distinction on the portion of the code of `eval` that computes the result of the step. The resoning is very similar to the proof of Lemma 3.

A.2 Proof of Theorem 1 (Trace soundness)

Proof. By induction on the number of `eval` steps using Lemmas 4 and 2. □

A.3 Auxiliary definitions and results for bound preservation

In order to prove the preservation of the bounds obtained in [1] we need to prove that for any trace $\rightarrow$ there is and equivalent trace using the semantics $\rightsquigarrow$ considered in [1]. These two semantics have some syntactic differences but they have the same behavior, so the correspondence is straightforward. In this case the correspondence is not one-to-one because the semantics $\rightsquigarrow$ has a rule to nondeterministically select the next process to execute in an object when it is idle—namely rule (11)—whereas our semantics selects automatically the next process in the queue when a process finishes or becomes blocked. Performing one $\rightarrow$-step can require two $\rightsquigarrow$-steps, but in that case the first one executes the same statement S as $\rightarrow$ and the second one does not execute any instruction (its decoration is ϵ). Therefore the statements executed will be the same in both semantic calculus.

The language presented in Section 2 and its semantics in Fig. 2 and 3 are a simplified version of those in [1]. The main differences are:

- the representation of the states
- the syntax of method invocations (both synchronous and asynchronous),
- the consideration of local variables and class declarations

In [1] states St are sets of futures and objects, which contain their queues of pending tasks. Formally an object is represented as $ob(o, C, h, \langle tv, \bar{b}\rangle, \mathcal{Q})$, where o is the *object identifier*, C is the *class*, h is the *object heap*, tv is the *table of local variables*, $\bar{b}$ is the sequence of instructions to execute, and $\mathcal{Q}$ the *set of pending tasks*. Futures are represented as $fut(fn, v)$, where f is the future identifier and v its value, possibly $\perp$. The operational semantics in [1] rewrites states $St \rightsquigarrow St'$.

We will consider a slight variation of the operational semantics in [1] where fields can be directly assigned by `new` and `get` instructions or arbitrary expression in the right-handd side, and future variables can be fields instead of local variables. This modification does not affect the upper bounds and the results obtained in [1]. To simplify the results, we will assume that the decorations of the $\rightsquigarrow$-steps use the syntax presented in Section 2.

In order to prove Theorem 2 we will define a translation from configurations as defined in Section 2.1 to states in the semantics in [1]. The translation will use the following functions, considering a configuration $\langle C, h\rangle$:

- $objs(C)$: returns the set of object identifiers in the set C.
- $futs(h)$: returns the set of future variables in the heap h.

We define two translations for runtime configurations: $\|\cdot\|$ from runtinme configurations $\langle C, h\rangle$ to states St, and $\langle\!\langle\cdot\rangle\!\rangle$ from runtime configurations (h, s) to states St.

Definition 1 (Translation of states).

$$\begin{aligned}\|\langle C,h\rangle\| &= \{ob(n,\_,h(n),a,t)|(n:Q)\in C,(a,t)=\|Q\|_q\}\cup\\ &\quad\{ob(o,\_,\epsilon,\epsilon,\emptyset)|o\in Dom(h)\setminus objs(C)\}\cup\\ &\quad\{fut(fn,v)|fn\in futs(h),h(fn)=v\}\end{aligned}$$

$$\begin{aligned}\|\epsilon\|_q &= (\epsilon,\emptyset)\\ \|(S;l)\cdot(S_1;l_1)\cdot\ldots\cdot(S_n;l_n)\|_q &= (\langle[\texttt{ret}\mapsto l],\|S\|_s\rangle,\\ &\quad\{\langle[\texttt{ret}\mapsto l_1],\|S_1\|_s\rangle,\ldots,\langle[\texttt{ret}\mapsto l_n],\|S_n\|_s\rangle\}\end{aligned}$$

$$\begin{aligned}\|\epsilon\|_s &= \epsilon\\ \|\texttt{x:=y};S\|_s &= \texttt{x:=y};\|S\|_s\\ \|\texttt{x:=r};S\|_s &= \texttt{x:=r};\|S\|_s\\ \|\texttt{x:=new};S\|_s &= \texttt{x:=new};\|S\|_s\\ \|\texttt{x:=f.get};S\|_s &= \texttt{x:=f.get};\|S\|_s\\ \|\texttt{f:=x!p}(\bar{z});S\|_s &= \texttt{call(m,p(x,}\bar{z}\texttt{,f))};\|S\|_s\\ \|\texttt{f:=p}(\bar{z});S\|_s &= \texttt{call(b,p(this,}\bar{z}\texttt{,\_))};\|S\|_s\\ \|\texttt{await f};S\|_s &= \texttt{await f};\|S\|_s\\ \|\texttt{return x};S\|_s &= \texttt{return x};\|S\|_s\end{aligned}$$

Definition 2 (Global translation). $\langle\!\langle(h,s)\rangle\!\rangle = \|^c[\![(h,s)]\!]^{-1}\|$

Finally we define the notion of *relevant* trace of $\rightsquigarrow$ steps, i.e., those that execute an statement.

Definition 3 (Relevant trace). *Given a trace* $\mathcal{T}_C = St_1 \rightsquigarrow^{o_1}_{S_1} St_2 \rightsquigarrow^{o_2}_{S_2} \ldots \rightsquigarrow^{o_{n-1}}_{S_{n-1}} St_n$ *we define the relevant trace of* $\mathcal{T}_C$ *as those steps that execute an statement:*

$$rel(\mathcal{T}_C) = \{St_i \rightsquigarrow^{o_i}_{S_i} St_{i+1} | St_i \rightsquigarrow^{o_i}_{S_i} St_{i+1} \in \mathcal{T}_C, S_i \neq \epsilon\}$$

Based on the equivalence between $\rightarrow$ and $\rightsquigarrow$ and Theorem 1 we can prove a resource preservation result wrt. $\rightsquigarrow$: for any sequence $\mathcal{T}_E$ of `eval` steps there is a corresponding trace $\mathcal{T}_C$ using the $\rightsquigarrow$ semantics from [1] *with the same cost.* We will use the translation function $\langle\!\langle\cdot\rangle\!\rangle$ to convert from runtime configurations (h,s) to the states in $\rightsquigarrow$.

Lemma 5 (Consumption Preservation wrt. $\rightsquigarrow$). *Let* (h_1,s_1) *be an initial state and consider a sequence* $\mathcal{T}_E$ *of* $n-1$ *consecutive* `eval` *steps defined as: a)* $o_i = nextObject(h_i,s_i)$*, b)* $\texttt{(res}_i\texttt{, l}_i\texttt{, h}_{i+1}\texttt{) = eval o}_i\ \texttt{h}_i$*, c)* $s_{i+1} = updL(s_i,o_i,l_i)$*. Then there is a trace* $\mathcal{T}_C = \langle\!\langle(h_1,s_1)\rangle\!\rangle \rightsquigarrow^* \langle\!\langle(h_n,s_n)\rangle\!\rangle$ *such that* $\mathcal{C}(\mathcal{T}_E,o,\mathcal{M}) = \mathcal{C}(\mathcal{T}_C,o,\mathcal{M})$.

Proof. By Theorem 1 we have that there is a trace (recall that $S_i \equiv res_i$)

$$\mathcal{T} = {}^c[\![(h_1,s_1)]\!]^{-1} \rightarrow^{o_1}_{S_1} {}^c[\![(h_2,s_2)]\!]^{-1} \rightarrow^{o_2}_{S_2} \ldots \rightarrow^{o_{n-1}}_{S_{n-1}} {}^c[\![(h_n,s_n)]\!]^{-1}$$

Since both traces execute the same statements in the same objects, then

$$\mathcal{C}(\mathcal{M},o,\mathcal{T}_E) = \mathcal{C}(\mathcal{M},o,\mathcal{T})$$

By Lemma 7 (see below) then there is a trace $\mathcal{T}_C = \|^c[\![(h_1, s_1)]\!]^{-1}\| \rightsquigarrow^* \|^c[\![(h_n, s_n)]\!]^{-1}\|$ such that

$$rel(\mathcal{T}_C) = \|^c[\![(h_1, s_1)]\!]^{-1}\| \rightsquigarrow_{S_1}^{o_1} \|^c[\![(h_2, s_2)]\!]^{-1}\| \rightsquigarrow_{S_2}^{o_2} \ldots \rightsquigarrow_{S_{n-1}}^{o_{n-1}} \|^c[\![(h_n, s_n)]\!]^{-1}\|$$

As before, $\mathcal{T}$ and $rel(\mathcal{T}_C)$ execute the same statements in the same objects, so

$$\mathcal{C}(\mathcal{M}, o, \mathcal{T}) = \mathcal{C}(\mathcal{M}, o, rel(\mathcal{T}_C))$$

By Lemma 8 (see below) the cost of a cost of $\mathcal{T}_C$ is the same as the cost of its relevant trace $rel(\mathcal{T}_c)$, so finally

$$\mathcal{C}(\mathcal{M}, o, \mathcal{T}_E) = \mathcal{C}(\mathcal{M}, o, \mathcal{T}_C)$$

□

Lemma 6. *If* $\langle C, h\rangle \rightarrow_b^n \langle C', h'\rangle$ *then:*

- $\|\langle C, h\rangle\| \rightsquigarrow_{\|b\|_s}^n \|\langle C', h'\rangle\|$ *or,*
- $\|\langle C, h\rangle\| \rightsquigarrow_{\|b\|_s}^n S \rightsquigarrow_\epsilon^n \|\langle C', h'\rangle\|$

Proof. By case distinction on the derivation applied to perform the $\rightarrow$-step.

- **(Internal)+(Assign I)**.

$$\text{(Internal)}\frac{\text{(Assign I)}\ \dfrac{h' = h[(n)(x) \mapsto h(n)(y)]}{\langle n : (\texttt{x:=y}; S, l) \cdot Q, h\rangle \rightarrow \langle n : (S, l) \cdot Q, h'\rangle}}{A \equiv \langle (n : (\texttt{x:=y}; S, l) \cdot Q) \cup C, h\rangle \rightarrow_{\texttt{x:=y}}^n \langle (n : (S, l) \cdot Q) \cup C, h'\rangle \equiv B}$$

  The translation of S_1 is

$$\|A\| = \{ob(n, \_, h(n), \langle [\texttt{ret} \mapsto l], \texttt{x:=y}; \|S\|_s\rangle, Q_{tr})|R\}$$

  where R is the rest of objects and future variables not involved in the step and Q_{tr} the translation of Q. From $\|A\|$ it is possible to perform a $\rightsquigarrow$-step using rule (2) in [1], reaching $\|B\|$:

$$(2)\frac{v = h(n)(y)}{\begin{array}{l}\{ob(n, \_, h(n), \langle [\texttt{ret} \mapsto l], \texttt{x:=y}; \|S\|_s\rangle, Q_{tr})|R\} \rightsquigarrow_{\texttt{x:=y}}^n \\ \{ob(n, \_, h(n)[x \mapsto v], \langle [\texttt{ret} \mapsto l], \|S\|_s\rangle, Q_{tr})|R\} \equiv \|B\|\end{array}}$$

- **(Internal)+(Assign II)**. Similar to the (Internal)+(Assign I) case.
- **(Internal)+(New)**.

$$\text{(Internal)}\frac{\text{(New)}\ \dfrac{\begin{array}{c}h(count) = m \\ h' = h[(n)(x) \mapsto m, (m) \mapsto \epsilon, count \mapsto m + 1]\end{array}}{\langle n : (\texttt{x:=new}; S, l) \cdot Q, h\rangle \rightarrow \langle n : (S, l) \cdot Q, h'\rangle}}{A \equiv \langle (n : (\texttt{x:=new}; S, l) \cdot Q) \cup C, h\rangle \rightarrow_{\texttt{x:=new}}^n \langle (n : (S, l) \cdot Q) \cup C, h'\rangle \equiv B}$$

The translation of S_1 is

$$\|A\| = \{ob(n, \_, h(n), \langle[\texttt{ret} \mapsto l], \texttt{x:=new}; \|S\|_s\rangle, Q_{tr})|R\}$$

From $\|S_1\|$ it is possible to perform a $\rightsquigarrow$-step using rule (3) in [1], reaching $\|B\|$:

$$(3)\ \frac{m = newRef() \qquad newHeap(\_, \epsilon)}{\begin{array}{l}\{ob(n, \_, h(n), \langle[\texttt{ret} \mapsto l], \texttt{x:=new}; \|S\|_s\rangle, Q_{tr})|R\} \rightsquigarrow^n_{\texttt{x:=new}} \\ \{ob(n, \_, h(n)[x \mapsto m], \langle[\texttt{ret} \mapsto l], \|S\|_s\rangle, Q_{tr}), \{ob(m, \_, \epsilon, \epsilon, \emptyset)|R\} = \|S_2\|\end{array}}$$

Note that m is a new object reference as it has been generated using the counter, and the heap of the new object generated by $newHeap$ is ϵ because we do not consider class declarations. No object with identifier m appears in C of B, but it is generated by the translation because m is in the domain of h (second set of $\|\cdot\|$)

– **(Internal)+(Get)**.

$$(\textsc{Internal})\ \frac{(\textsc{Get})\ \dfrac{h(h(n)(f)) \neq \bot \qquad h' = h[(n)(x) \mapsto h(h(n)(f))]}{\langle n : (\texttt{x:=f.get}; S, l) \cdot Q, h\rangle \rightarrow \langle n : (S, l) \cdot Q, h'\rangle}}{\begin{array}{l}A \equiv \langle (n : (\texttt{x:=f.get}; S, l) \cdot Q) \cup C, h\rangle \rightarrow^n_{\texttt{x:=f.get}} \\ \langle (n : (S, l) \cdot Q) \cup C, h'\rangle \equiv B\end{array}}$$

The translation of A is:

$$\|A\| = \{ob(n, \_, h(n), \langle[\texttt{ret} \mapsto l], \texttt{x:=f.get}; \|S\|_s\rangle, Q_{tr}), fut(fn, v)|R\}$$

From $\|A\|$ it is possible to perform a $\rightsquigarrow$-step using rule (8) in [1]:

$$(8)\ \frac{h(n)(f) = fn \qquad v \neq \bot}{\begin{array}{l}\{ob(n, \_, h(n), \langle[\texttt{ret} \mapsto l], \texttt{x:=f.get}; \|S\|_s\rangle, Q_{tr}), fut(fn, v)|R\} \rightsquigarrow^n_{\texttt{x:=f.get}} \\ \{ob(n, \_, h(n)[x \mapsto v], \langle[\texttt{ret} \mapsto l], \|S\|_s\rangle, Q_{tr}), fut(fn, v)|R\} = \|B\|\end{array}}$$

Note that by the definition of the translation $\|\cdot\|$ we have that $h(h(n)(f)) = v$

– **(Internal)+(Await I)**.

$$(\textsc{Internal})\ \frac{(\textsc{Await I})\ \dfrac{h(h(n)(f)) \neq \bot}{\langle n : (\texttt{await f}; S, l) \cdot Q, h\rangle \rightarrow \langle n : (S, l) \cdot Q, h\rangle}}{\begin{array}{l}A \equiv \langle (n : (\texttt{await f}; S, l) \cdot Q) \cup C, h\rangle \rightarrow^n_{\texttt{await f}} \\ \langle (n : (S, l) \cdot Q) \cup C, h\rangle \equiv B\end{array}}$$

The translation of A is:

$$\|A\| = \{ob(n, \_, h(n), \langle[\texttt{ret} \mapsto l], \texttt{await f}; \|S\|_s\rangle, Q_{tr}), fut(fn, v)|R\}$$

From $\|A\|$ it is possible to perform a $\rightsquigarrow$-step using rule (9) in [1]:

$$(9)\ \frac{h(h(n)(f)) \neq \bot}{\begin{array}{l}\{ob(n, \_, h(n), \langle[\texttt{ret} \mapsto l], \texttt{await f}; \|S\|_s\rangle, Q_{tr}), fut(fn, v)|R\} \rightsquigarrow^n_{\texttt{await f}} \\ \{ob(n, \_, h(n), \langle[\texttt{ret} \mapsto l], \|S\|_s\rangle, Q_{tr}), fut(fn, v)|R\} = \|B\|\end{array}}$$

Note that by the definition of the translation $\|\cdot\|$ we have that $h(n)(f) = fn$.

– **(Internal)+(Await II)**. This case is similar to the previous one but possibly involving 2 $\rightsquigarrow$-steps: one that evaluates the `await f` that cannot continue and releases the object, and one that schedules the next task in the object.

$$\text{(Internal)}\ \frac{\text{(Await II)}\ \dfrac{h(h(n)(f)) = \bot}{\langle n : (\texttt{await f}; S, l) \cdot Q, h\rangle \to \langle n : Q \cdot ((\texttt{await f}; S, l)), h\rangle}}{\begin{array}{l} A \equiv \langle (n : (\texttt{await f}; S, l) \cdot Q) \cup C, h\rangle \to^{n}_{\texttt{await f}} \\ \langle (n : Q \cdot ((\texttt{await f}; S, l))) \cup C, h\rangle \equiv B \end{array}}$$

Consider that $\|Q \cdot ((\texttt{await f}; S, l))\|_q = (a, t)$, where a is the translation of the first task in the queue and t the translation of the rest of the queue. The translation of A is:

$$\|A\| = \{ob(n, \_, h(n), \langle[\texttt{ret} \mapsto l], \texttt{await f}; \|S\|_s\rangle, Q_{tr}), fut(fn, v)|R\}$$

From $\|A\|$ we can perform a $\rightsquigarrow$-step using rule (10) in [1]:

$$(10)\ \frac{h(h(n)(f)) = \bot}{\begin{array}{l} \{ob(n, \_, h(n), \langle[\texttt{ret} \mapsto l], \texttt{await f}; \|S\|_s\rangle, Q_{tr}), fut(fn, v)|R\} \rightsquigarrow^{n}_{\texttt{await f}} \\ \{ob(n, \_, h(n), \epsilon, \langle[\texttt{ret} \mapsto l], \texttt{await f}; \|S\|_s\rangle \cup Q_{tr}), fut(fn, v)|R\} = A' \end{array}}$$

Similar to the previous case, we know that $h(n)(f) = fn$. Then from the state A' we can apply rule (11) to schedule the first task a in the queue:

$$(11)\ \frac{a \in \langle[\texttt{ret} \mapsto l], \texttt{await f}; \|S\|_s\rangle \cup Q_{tr}}{\begin{array}{l} \{ob(n, \_, h(n), \epsilon, \langle[\texttt{ret} \mapsto l], \texttt{await f}; \|S\|_s\rangle \cup Q_{tr}), fut(fn, v)|R\} \rightsquigarrow^{n}_{\epsilon} \\ \{ob(n, \_, h(n), a, t), fut(fn, v)|R\} = \|B\| \end{array}}$$

Therefore we have the two-step $\rightsquigarrow$-derivation $\|A\| \rightsquigarrow^{n}_{\texttt{await f}} A' \rightsquigarrow^{n}_{\epsilon} \|B\|$.

– **(Internal)+(Sync)**.

$$\text{(Internal)}\ \frac{\text{(Sync)}\ \dfrac{(m(\bar{w}) \mapsto S_m) \in D\ \mathit{fresh} \qquad \tau = [\bar{w} \mapsto h(n)(\bar{z})] \qquad S' = \widehat{(S_m\tau)}^{x}}{\langle n : (\texttt{x:=m}(\bar{z}); S, l) \cdot Q, h\rangle \to \langle n : (S'; S, l) \cdot Q, h\rangle}}{S_1 \equiv \langle (n : (\texttt{x:=m}(\bar{z}); S, l) \cdot Q) \cup C, h\rangle \to^{n}_{\texttt{x:=m}(\bar{z})} \langle (n : (S'; S, l) \cdot Q) \cup C, h\rangle \equiv S_2}$$

The translation of S_1 is

$$\|S_1\| = \{ob(n, \_, h(n), \langle[\texttt{ret} \mapsto l], \texttt{call(b,m(this,}\bar{z}\texttt{,\_))}; \|S\|_s\rangle, Q_{tr})|R\}$$

where R is the rest of objects and future variables not involved in the step and Q_{tr} the translation of Q. From $\|S_1\|$ it is possible to perform a $\rightsquigarrow$-step using rule (4) in [1], reaching $\|S_2\|$:

$$(4)\ \frac{(m(\bar{w}) \mapsto S_m) \in \|D\|^{x}_{sync}\ \mathit{fresh} \qquad \tau = [\bar{w} \mapsto h(n)(\bar{z})]}{\begin{array}{l} \{ob(n, \_, h(n), \langle[\texttt{ret} \mapsto l], \texttt{call(b,m(this,}\bar{z}\texttt{,\_))}; \|S\|_s\rangle, Q_{tr})|R\} \rightsquigarrow^{n}_{\texttt{m}(\bar{z})} \\ \{ob(n, \_, h(n), \langle[\texttt{ret} \mapsto l], S_m\tau; \|S\|_s\rangle, Q_{tr})|R\} \equiv \|S_2\| \end{array}}$$

$\|D\|^x_{sync}$ is the translation of all the methods in the program D where methods are treated synchronously, i.e., they store a final value in the field x. We consider a simplification of the operational semantics in [1] where synchronous methods return exactly one value, thus the last instruction of a synchronous method stores the final value in the corresponding field. In this case it is easy to check that $\|\widehat{(S_m\tau)}^x\| = S_m\tau$.

- **(Message)+(Async)**. Similar to the previous case.
- **(Internal)+(Return$_A$)**.

$$\text{(Internal)}\ \frac{\text{(Return}_A\text{)}\ \dfrac{h' = h[(l) \mapsto h(n)(x)]}{\langle n : (\texttt{return x}; S, l) \cdot Q, h\rangle \to \langle n : Q, h'\rangle}}{A \equiv \langle (n : (\texttt{return x}; S, l) \cdot Q) \cup C, h\rangle \to^n_{\texttt{return x}} \langle (n : Q) \cup C, h'\rangle \equiv B}$$

The translation of A is

$$\|A\| = \{ob(n, \_, h(n), \langle [\texttt{ret} \mapsto l], \texttt{return x}; \|S\|_s\rangle, Q_{tr}), fut(l, \bot) | R\}$$

where R is the rest of objects and future variables not involved in the step and Q_{tr} the translation of Q. From $\|A\|$ it is possible to perform a $\rightsquigarrow$-step using rule (7) in [1]:

$$(7)\ \frac{v = h(n)(x)}{\begin{array}{l}\|A\| = \{ob(n, \_, h(n), \langle [\texttt{ret} \mapsto l], \texttt{return x}; \|S\|_s\rangle, Q_{tr}), fut(l, \bot) | R\} \rightsquigarrow^n_{\texttt{return x}} \\ \{ob(n, \_, h(n), \epsilon, Q_{tr}), fut(l, v) | R\} = A'\end{array}}$$

If $Q_{tr} = \epsilon$, i.e., if the process queue of object n is empty then we are done because $A' = \|B\|$. Otherwise we need to apply a step with rule (11) to select the next proces in the queue, performing a step $A' \rightsquigarrow^n_\epsilon \|B\|$ similar to the case (Await II)

- **(Internal)+(Return$_S$)**. Similar to the previous case (Return$_A$), but applying rule (6) instead of (7) in the $\rightsquigarrow$-step.

□

Lemma 7. *If $\mathcal{T} = A_1 \to^{o_1}_{S_1} A_2 \to^{o_2}_{S_2} \ldots \to^{o_{n-1}}_{S_{n-1}} A_n$ then there is a trace $\mathcal{T}_C = \|A_1\| \rightsquigarrow^* \|A_n\|$ such that $rel(\mathcal{T}_C) = \|A_1\| \rightsquigarrow^{o_1}_{S_1} \|A_2\| \rightsquigarrow^{o_2}_{S_2} \ldots \rightsquigarrow^{o_{n-1}}_{S_{n-1}} \|A_n\|$.*

Proof. Strightforward by induction on the number of steps in the trace $\mathcal{T}$, and applying Lemma 6. □

Lemma 8. *For any trace $\mathcal{T}_C$ wrt. $\rightsquigarrow$, cost model $\mathcal{M}$ and object reference o then $\mathcal{C}(\mathcal{T}_C, o, \mathcal{M}) = \mathcal{C}(rel(\mathcal{T}_C), o, \mathcal{M})$.*

Proof. By definition of the cost of trace (Definition 3 in [1]), since only the steps decorated with a statement (i.e., different from ϵ) contribute to the cost. □

A.4 Proof of Theorem 2 (Bound Preservation)

Proof. Straighforward by Lemma 5 and Theorem 3 from [1].

A.4 Benchmarking the ABS backends

A.4.1 Setup

```
Hardware: Intel i7-3537U (2 cores, 4 hyperthreads), 8GB RAM, Linux-64bit
The Glorious Glasgow Haskell Compilation System, version 7.10.1
ABS Tool Suite v1.2.3.201509291051-c6f3df1
OpenJDK Runtime Environment (build 1.8.0_60-b24) OpenJDK 64-Bit Server VM (build 25.60-b23, mixed mode)
Erlang/OTP 18 [erts-7.0] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe] [kernel-poll:false]
Maude 2.6 built: Dec 9 2010 18:28:39
```

A.4.2 Results

Haskell

Program	Time(s)	Cpu Utiliz.(%)	Memory(KB)	÷haskell-time	÷haskell-mem
BinarySearchTree	0.01	75	3584	1.00x	1.00x
FieldFutures	0.39	99	93200	1.00x	1.00x
NaiveFib	0.11	97	3900	1.00x	1.00x
Rosetree	0.01	0	3428	1.00x	1.00x
SumList	0.01	86	12020	1.00x	1.00x
ThreadRingLocal	0.06	96	4236	1.00x	1.00x
AwaitOnField	0.09	105	6348	1.00x	1.00x
AwaitOnFut	0.05	104	6132	1.00x	1.00x
Bang	0.22	136	10220	1.00x	1.00x
BenchLists	5.84	138	15320	1.00x	1.00x
BenchMaps	0.05	124	10408	1.00x	1.00x
Big	0.03	136	12964	1.00x	1.00x
Sequences	0.02	162	13376	1.00x	1.00x
SerialMsg	0.04	153	6256	1.00x	1.00x
StressTest	0.04	128	9952	1.00x	1.00x
SyncAsync	0.05	141	9360	1.00x	1.00x
ThreadRingCOG	0.2	136	9980	1.00x	1.00x

NewJava

Program	Time(s)	Cpu Utiliz.(%)	Memory(KB)	÷haskell-time	÷haskell-mem
BinarySearchTree	err	err	err	err	err
FieldFutures	timeout	timeout	timeout	timeout	timeout
NaiveFib	4.43	167	137756	40.27x	35.32x
Rosetree	err	err	err	err	err
SumList	err	err	err	err	err
ThreadRingLocal	0.2	186	42856	3.33x	10.12x
AwaitOnField	2.39	241	146276	26.56x	23.04x
AwaitOnFut	2.3	300	143580	46.00x	23.41x
Bang	0.8	289	151696	3.64x	14.84x
BenchLists	4.23	114.00	810860	0.72x	52.93x
BenchMaps	0.22	207	49100	4.40x	4.72x
Big	0.2	173	43016	6.67x	3.32x
Sequences	0.31	240	69420	15.50x	5.19x
SerialMsg	0.2	189	45320	5.00x	7.24x
StressTest	0.57	306	104144	14.25x	10.46x
SyncAsync	0.2	166	44584	4.00x	4.76x
ThreadRingCOG	0.2	173	42752	1.00x	4.28x

OldJava

Program	Time(s)	Cpu Utiliz.(%)	Memory(KB)	÷haskell-time	÷haskell-mem
BinarySearchTree	0.31	198	73340	31.00x	20.46x
FieldFutures	out-of-mem	out-of-mem	out-of-mem	out-of-mem	out-of-mem
NaiveFib	16.56	123	658188	150.55x	168.77x
Rosetree	0.14	149	54564	14.00x	15.92x
SumList	1.11	300	192328	111.00x	16.00x
ThreadRingLocal	7.35	223	830612	122.50x	196.08x
AwaitOnField	7.1	136	487328	78.89x	76.77x
AwaitOnFut	7.82	141	354432	156.40x	57.80x
Bang	5.96	235	755740	27.09x	73.95x
BenchLists	86.78	389	792784	14.86x	51.75x
BenchMaps	96.66	391	796512	1933.20x	76.53x
Big	6.95	175	1172832	231.67x	90.47x
Sequences	8.74	182	788196	437.00x	58.93x
SerialMsg	3.01	263	803224	75.25x	128.39x
StressTest	2.53	205	797100	63.25x	80.09x
SyncAsync	23	144	1183192	460.00x	126.41x
ThreadRingCOG	38.07	186	1099024	190.35x	110.12x

Erlang

Program	Time(s)	Cpu Utiliz.(%)	Memory(KB)	÷haskell-time	÷haskell-mem
BinarySearchTree	1.28	22	22824	128.00x	6.37x
FieldFutures	timeout	timeout	timeout	timeout	timeout
NaiveFib	1.63	39	24796	14.82x	6.36x
Rosetree	1.24	19	22688	124.00x	6.62x
SumList	1.29	23	40716	129.00x	3.39x
ThreadRingLocal	timeout	timeout	timeout	timeout	timeout
AwaitOnField	1.65	78	30492	18.33x	4.80x
AwaitOnFut	1.91	76	26184	38.20x	4.27x
Bang	422.23	133	261776	1919.23x	25.61x
BenchLists	58.67	330	371596	10.05x	24.26x
BenchMaps	56.97	336	428192	1139.40x	41.14x
Big	21.51	230	654056	717.00x	50.45x
Sequences	44.01	207	34036	2200.50x	2.54x
SerialMsg	timeout	timeout	timeout	timeout	timeout
StressTest	8.3	252	75216	207.50x	7.56x
SyncAsync	13.97	287	377516	279.40x	40.33x
ThreadRingCOG	284.71	311	166128	1423.55x	16.65x

Maude

Program	Time(s)	Cpu Utiliz.(%)	Memory(KB)	÷haskell-time	÷haskell-mem
BinarySearchTree	0.5	99	47896	50.00x	13.36x
FieldFutures	timeout	timeout	timeout	timeout	timeout
NaiveFib	198.62	100	38724	1805.64x	9.93x
Rosetree	0.29	98	39444	29.00x	11.51x
SumList	timeout	timeout	timeout	timeout	timeout
ThreadRingLocal	timeout	timeout	timeout	timeout	timeout
AwaitOnField	320.21	100	41672	3557.89x	6.56x
AwaitOnFut	328.65	100	41908	6573.00x	6.83x
Bang	timeout	timeout	timeout	timeout	timeout
BenchLists	timeout	timeout	timeout	timeout	timeout
BenchMaps	timeout	timeout	timeout	timeout	timeout
Big	timeout	timeout	timeout	timeout	timeout
Sequences	timeout	timeout	timeout	timeout	timeout
SerialMsg	timeout	timeout	timeout	timeout	timeout
StressTest	timeout	timeout	timeout	timeout	timeout
SyncAsync	timeout	timeout	timeout	timeout	timeout
ThreadRingCOG	timeout	timeout	timeout	timeout	timeout