Universiteit
Leiden
The Netherlands

Abstract Behavioral Specification: unifying modeling and programming

Bezirgiannis, N.

Citation

Bezirgiannis, N. (2018, April 17). *Abstract Behavioral Specification: unifying modeling and programming*. *IPA Dissertation Series*. Retrieved from https://hdl.handle.net/1887/61629

Version:	Not Applicable (or Unknown)
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/61629

Note: To cite this publication please use the final published version (if applicable).

Cover Page

Universiteit Leiden

The following handle holds various files of this Leiden University dissertation:
http://hdl.handle.net/1887/61629

Author: Bezirgiannis, N.
Title: Abstract Behavioral Specification: unifying modeling and programming
Issue Date: 2018-04-17

Chapter 3

HABS: A Variant of the ABS Language

The background text on chapter 2 covered the basic characteristics of the ABS language, which we name *Standard ABS*; however, ABS can be better regarded as a *family* of languages. Indeed, there are different variations (in terms of omissions and extensions) to the Standard ABS, each focusing on specific goals, e.g. on completeness of semantics (Maude-ABS [Johnsen et al., 2010a]), correctness (KeY-ABS [Din et al., 2015]), model-checking (Maude-ABS), simulation (Erlang-ABS [Göri et al., 2014]), etc.. The variant of Standard ABS that is described in this chapter focuses instead on performance of execution and is given the name *HABS* (short version of Haskell-ABS), since it is implemented on top of the Haskell language & runtime.

3.1 Differences with Standard ABS

Most features of Standard ABS are supported by HABS. We discuss in this section explicitly their differences and deviations. Standard ABS, like Java, allows the re-assignment of passed method parameters, as in the example ABS code:

```
class C {
  Unit method(Int x) {
    x = 3; // reassignment of method parameter
  }
}
```

However, HABS disallows such re-assignments for two reasons: first, it is considered bad programming practice to re-assign method parameters since it leads to confusion over how the parameters are passed (call-by-value or call-by-reference) and secondly, the parallel and, more importantly, the distributed implementation of ABS become faster and straightforward. For reference, the OO mainstream languages Scala and OCaml also disallow such re-assignments of method parameters. Going even further, HABS disallows the re-assignment of captured patterns in case-statements. There is no such issue for the case-expression since identifiers inside functional code cannot be mutated, but only "shadowed". An example of the two different cases:

```
{
 case (3) { // case−statement
    x => {
          x = x+1; // reassignment
           println (toString(x));
          }
 }

 println (case (3) { // case−expression
             y => let (Int y) = y + 1 // shadowing
                  in toString(y);
                   });
}
```

A way to overcome this restriction of re-assignment for both method parameters and case-statement patterns, is to manually rename the formal parameter of method and assign it to a (re-assignable) variable in the beginning of the method's body, as in the example:

```
class C {
  Unit method(Int renamed_x) {
    Int x = renamed_x; // extra assignment
    x = x + 1; // rest of code remains the same
          println (toString(x));
         }
 }
```

Continuing on, Standard ABS does not define any default ordering of objects and futures; as such, the various ABS implementations implement differently this ordering which may be stable or not across the whole program

execution or even across multiple same-program executions. Because of this, HABS decides to not provide at all any default ordering (via the builtin comparison operators >,<,<=,>=)) for objects and futures. The reason for not providing such a *default* ordering is twofold. 1) There is no agreed notion of what the ordering should be for objects and futures: is it structural (natural) ordering or physical ordering (e.g. depending on creation time or memory-address allocation)? 2) An implementation of ordering adds certain overhead (for tagging the data), especially in the case where stable ordering is required, over one program execution or even worse over multiple program executions — any non-determinism of the program would then have to be eliminated. The OO mainstream language Java also does not provide such default object and future ordering but instead forces the user to manually provide it, by implementing the Comparable.comparesTo() method.

This HABS restriction poses a limitation when objects have to appear in the (fast) Set abstract datatype or as keys of the (fast) Map abstract datatype, which are provided by the ABS Standard Library. A workaround at the moment for Set is the choice to use a slow implementation in the Standard Library (one that does not depend on ordering of elements); for the case of Map the HABS user has to do manual tagging.

Futures, as described in Standard ABS of section 2.7 are write-once containers of values. As such they could be covariantly subtyped (see section 2.4.3). Indeed, certain ABS backends (Erlang-ABS, Maude-ABS) allow for futures to be covariant; however, for implementation reasons (relating to Haskell) futures in HABS are not covariant but invariant, i.e. their contained type cannot change. This does not happen to be a big issue in practice since covariance can be achieved by extracting the contained value (via future.get), as the example:

```
{
  Fut<Int> f = object!method();
  Fut<Rat> f_ = f; // type error for HABS, ok for other backends

  Int v = f.get;
  Rat v_ = v; // ok for HABS and other backends
}
```

The above workaround is not applied automatically for reasons of efficiency. HABS has limited support for fields pointing to futures. Specifically, consider the ABS example of a future-field:

```
1 class C {
2  Fut<A> f; // A future field
3
```

```
4  Unit method() {
5    await this.f?;
6  }
7   ...
8  }
```

The await of Line 5 says that the current execution has to yield control at least until the future pointed by this.f is resolved. In other words, the future that is stored *at the moment of resumption* inside the field f must be completed. This means that any standard-ABS backend must not only track for the completion of the future, but also for any modifications to fields that contain futures. For performance reason, HABS does not currently track any modifications to future-fields: this means that the execution will be resumed when the future that was pointing at the first time of evaluating the statement await at Line 5, and regardless of any modifications happened to the field in the meantime of the suspension. This restriction leads to different semantics of future-fields compared to Standard ABS and as such may yield to deadlocks that would not occur otherwise.

Compared to other ABS backends, HABS disallows certain "effectful" expressions of the ABS Standard Library (e.g. random,print, println , readln) to be placed inside pure functional code. This can be considered not a limitation but actually an advantage, since HABS strictly and safely separates functional code from any side-effectful ABS code.

Finally, there is currently no standardization of how any ABS datum (primitive, ADT, object) is textually represented (via the toString() function). Consequently, there is no serialization format proposed for ABS data types. HABS employs its own textual representation for ABS data, which may differ from other ABS language implementations.

3.2 Language extensions to Standard ABS

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 for Java) with new syntactic and semantic support for interfacing to Haskell libraries.

3.2.1 Exceptions

A feature that was previously lacking and recently added to the ABS language is the capability to signal program faults and recover from them. This language

extension came as a prerequisite to the support for real-world deployments of ABS software. Faults commonly appear in real-world systems, especially in distributed settings. Therefore, a robust mechanism in the form of exceptions was designed in place.

As a starting point for adding exceptions to ABS, the project undertook a survey of the design space; a summary can be found in [Lanese et al., 2014]. This section describes the extension that was subsequently implemented.

To be compatible with the functional core of the language, the exception type is modelled as an Algebraic Data Type (ADT). A single *open* data type is introduced with the name `Exception`. The programmer can extend this basic data type by augmenting it with user-specific exceptions (data constructors). The ABS standard library also comes bundled with certain predefined system-level exceptions (see table 3.1); note that the number of predefined exceptions may differ between ABS backends. The language, however, makes no distinction between system and user exceptions, synchronous and asynchronous exceptions. Synchronous exceptions are mostly user-level written exceptions, where their occurrence can be traced back to the original program code (e.g. a call to throw); as such, synchronous exceptions can happen only in specific program points. Asynchronous exceptions, on the other hand, can happen anywhere in the program and their occurrence cannot be traced back to an explicit call to throw; most of these exceptions are generated by the system, e.g. in other languages `StackOverflowException`, `OutOfMemoryException`, `ThreadKilledException`. Exceptions in ABS, similar to ADTs, take 0 or more arguments as exemplified:

```
exception MyException;
exception AnotherException(Int, String, Bool);
```

Furthermore, the language treats exceptions as first-class citizens; the user can construct exception-values, assign them to variables or pass them in expressions. An exception can be explicitly raised with the `throw` statement as:

```
{
  throw AnotherException(3, "mplo");
}
```

When an exception is raised the normal flow of the program will be aborted. In order to resume execution in the current process, the user has to explicitly *handle* the exception. This is achieved with a try−catch−finally compound statement similar to Java, with the only difference being that the user can pattern-match on each catch-clause for the exception-constructor arguments.

DivisionByZeroException	Automatically thrown from expressions that evaluate to x/0
PatternMatchFailException	No pattern matched in case or catch clause, and there was no wildcard (_) pattern.
AccessorException	Applied data accessor does not match input data value
AssertionFailException	Argument to assert is False
NullPointerException	Method call on a null object

Table 3.1: Predefined exceptions for HABS Standard Library

Statements in the try block will be executed and upon a raised exception the flow of execution will be transferred to the catch block, so as to handle (catch) the exception.

The catch block behaves similar to the case statement, although the patterns inside a catch block can only have the type Exception. Every such pattern is tried in order and if there is a match, its associated statements will be executed.

The catch block is followed by an optional finally block of statements, that will be executed regardless of an exception happening or not. The syntax is the following:

```
try {
 stmt1;
 stmt2;
  ....
}
catch {
   exception_pattern1   => stmt_or_block;
   exception_pattern2   => ... ;
   ...
   _ => ...
}
 finally  {
  stmt3;
  stmt4;
}
```

In case there is no matching exception pattern, the optional finally block will be executed and the exception will be propagated in turn to the parent

caller, and so forth, until a match is made. In the case that the propagation reaches the top-caller in the process call-stack without a successful catch, the process will be abruptly exited. Processes that were waiting on the future of the exited process will be notified with a ProcessExitedException.

The associated object where the exited process was operating on will remain live. That means, all other processes of the same object will not be affected. There is, however, a special exception case (named `die`) in the distributed version of ABS (see section 5.1.4) where the object and all of its processes are also exited.

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.

3.2.2 Parametric type synonyms

As shown in section 2.4.4, Standard ABS supports only "plain" type synonyms, which can be thought of as aliases, assigning a (shorter) type name to another (possibly "longer") type name; this is similar to Go's language version 1.9 type-aliases feature. Going a step further, the HABS implementation supports more expressive type synonyms, which are so-called *parametric type synonyms*. As the name suggests, such synonyms can take parameters, i.e. type variables, which allows to combine type aliasing with parametric polymorphism. An example of a common parametric type synonym in the functional world is the Error type: "functional" errors can be thought of chains of computations that may abruptly throw an error — in our simple case, the error is represented as a String which textually describes what occurred — or complete successfully with a result. These two choices can be implemented by the sum type (Either) where by convention Left represents the erroneous situation and Right the successful computation, e.g. in HABS:

```
type Error<A> = Either<String,A>;
```

In Standard ABS, instead of HABS, we could not supply such parameter A so we can be abstract over all result types. In HABS, the parametric type synonyms can be further nested, e.g.:

```
type WorkFlow<A> = Pair<Iterations, List<Error<A>>>;
type Iterations  = Int
```

3.2.3 Type Inference

We extend the syntax and type system of ABS to allow type inference. The user adds a wildcard and the underlying type checker will try to infer its type, as in the HABS example:

```
{
  _ name = "MyName";
  Map<_,Int> salaries = insert(emptyMap(),Pair(name,30000));
}
```

The wildcard here will be replaced by the typechecker ("inferred") by `String`. These partial type signatures are influenced by the recent Haskell `PartialTypeSignatures` language pragma extension. Similar to Haskell's type inference, the HABS type inference is not complete: particularly, types that are governed by nominal subtyping rules (i.e. interface types) may fail to be inferred by the HABS compiler.

3.2.4 Foreign Language Interface

The Standard ABS did not define any interface to a foreign language. However, based on the demand by modellers for having a library of efficient datastructures (e.g. arrays, hashtables), the previously most popular ABS backend named Java-ABS backend (to distinguish from the newer Java8-ABS backend) added a Foreign Language Interface (FLI) to the ABS language, by means of reflection, ABS annotations and class stubs. More specifically, a Java-ABS user has to add the `Foreign` annotation on any ABS class that should be implemented by foreign code, as in the example (taken from the Java-ABS repository):

```
import Foreign from ABS.FLI;

interface Random {
    Int random(Int max); // Generate random integer between (0, max]
}

[Foreign]
class Random implements Random { // STUB class
    Int random(Int max) { // this method is overridden by java
```

```
        return max;
    }
}

{
    Random rnd = new local Random();
    Int n = rnd.random(100);
}
```

This Random foreign class is a short of a code stub: the ABS user can however provide with a default implementation in ABS (e.g. a dummy value here of max), in case there is no support for a particular foreign language or the code is supposed to run with a different backend that lacks this FLI extension (i.e. any other backend). Although, the Java-ABS backend did not declare any restrictions on what foreign languages are supported, there exists one implementation of only interfacing with Java code. The following Java snippet is a class that overrides the ABS Random class — some naming conventions are assumed.

```
package Env;

import abs.backend.java.lib.types.*;
import java.util.Random;

public class Random_fli extends Random_c {

    public ABSInteger fli_random(ABSInteger max) {
        Random rnd = new Random();
        int n = rnd.nextInt(max.toInt());
        return ABSInteger.fromInt(n);
    }
}
```

Although this approach of Java-ABS keeps the ABS codebase compatible with other ABS backends, it limits the support for foreign languages only to those that admit to object-oriented paradigm, since it relies on subclassing. Since our goal is to use the Haskell runtime — Haskell lacks OO — and driven also by the observation that most mainstream languages want to interface to lower-level code (and thus not OO), for example C, we devised a new extension to the ABS language that is not OO-bound. This foreign language interface for HABS was designed around the ABS module system. The user has to simply prefix an import declaration with `foreign`. This new syntax directive is shown in the example:

```
// For generating random numbers
foreign import GenIO from System.Random.MWC;
foreign import createSystemRandom from System.Random.MWC;
foreign import uniformR from System.Random.MWC;

{
  GenIO g = createSystemRandom();
  Int source = uniformR(Pair(1,100), g);
}
```

Here we import the GenIO random-generator datatype and associated procedures to create and roll a uniformly-distributed random number from the implementation of the `mwc-random` Haskell library. We can then use the imported procedures in ABS functions or statements as usual. Note that we did not define any types for the imported identifiers. As such, this FLI extension can be regarded as untyped: the ABS type checker does not do any prior typechecking, but assumes that the ABS user does the right thing (i.e. well-typing and not mixing functional with stateful code). In reality, an external typechecker of the foreign language could be applied for this reason. A further addition to this FLI extension which has not been implemented yet is adding static type support by extra type signatures, e.g. :

```
fimport quot from Prelude;

def Int quot(Int a, Int b) = foreign;
```

3.2.5 Language extension for HTTP communication

Finally, since ABS was primarily designed as a modeling language, it lacks the common I/O functionality found in mainstream programming languages. To allow user interaction, a new language extension was introduced built around an HTTP API. The ABS user may annotate any object declaration with [HTTPName: strExp()] I o = new ... to make the object and its fields accessible from the outside as an HTTP endpoint. Any such object can have some of its method definitions annotated with [HTTPCallable] to allow them to be called from the outside; the arguments passed and the method's result will be serialized according to a standard JSON format.

The HTTP API extension of ABS utilizes WARP: a high-performance, high-throughput server library written in Haskell. It is worth noting that any exposed objects (by using HTTPName) will not be processed by the Haskell's

garbage collector, and as such their lifespan reaches that of the whole ABS program. A snippet utilizing the HTTP-API extension follows, taken from the ABS Fredhopper case study (described at section 4.5):

```
interface Monitor {
  Maybe<ScaleStamp> monitor();
  [HTTPCallable] List<Pair<Time, List<Pair<String, Rat>>>> metricHistory();
}
interface MonitoringQueryEndpoint extends EndPoint {
  [HTTPCallable] Unit invokeWithDelay(Int proctime, String customer,
                                      Int amazonECU, Int delay);
}
{
  ...
  [HTTPName: "monitoringService"] MonitoringService ms=new MonitoringServiceImpl();
  [HTTPName : "monitor"] DegradationMonitorIf degradationMonitor =
                    new DegradationMonitorImpl(deployerif);

  Fut<Unit> df = ms!addMS(Rule_( 5000, degradationMonitor ));
  df.get;

  [HTTPName : "queryService"] MonitoringQueryEndpoint mqep = new
      MonitoringQueryEndpointImpl(loadBalancerEndPointsUs, degradationMonitor);
  println ("Endpoints set up. Waiting for requests ...");
}
```

3.3 Compiling ABS to Haskell

In this section, we introduce another backend approach. 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 performance 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

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

tool for our HABS backend, which was later adopted also by the Java8-ABS backend.

When starting off the HABS backend, the initial motivation was to develop a backend that can generate more efficient executable code compared to the markedly slower at the time Maude-ABS and 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 layer and subtype polymorphism that remained a particular challenge (see Section 3.3). 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".

The original Haskell backend of ABS was designed with performance in mind, as well as to offer distributed computing on the cloud [Bezirgiannis and Boer, 2016]. Algebraic-datatypes, parametric polymorphism, interfaces, pure functions are all one-to-one mapped down to Haskell. Haskell's type system lacks subtyping polymorphism, and as such we implement this in the HABS compiler itself through means of implicit coercive subtyping.

3.3.1 Compiler infrastructure

The HABS implementation of ABS translates ABS source to equivalent Haskell source (i.e. source-to-source compilation, also called transcompilation). We make use of BNFC converter `http://bnfc.digitalgrammars.com/` : a compiler generator which generates a fast parser written in Haskell from a BNF grammar that describes ABS. The HABS transcompiler, which is written in Haskell itself, translates input ABS abstract syntax tree to a Haskell abstract syntax tree in the output, which gets subsequently compiled by a Haskell compiler. We currently generate code that can only be compiled by the Glasgow Haskell Compiler (GHC), which is the most widely-used Haskell implementation.

The translation is mostly straightforward since the ABS and Haskell languages share certain similarities. The source code and installation instructions of the HABS transcompiler is located at `https://github.com/abstools/`

`habs`.

3.3.2 Functional code

At their core, the two languages, ABS and Haskell, are more or less the same, i.e. purely-functional languages with support for Algebraic Datatypes and parametric-polymorphism.

Pure functions and case-pattern matching of ABS are translated to the Haskell equivalents. The let construct of ABS (e.g. let (T x) = exp1 in exp2) is translated to a lambda abstraction plus its function application, that is (\ x -> exp2) (exp1::T). The reason that we can simply use lambdas for translation is that the let in ABS is monomorphic and non-recursive, unlike Haskell's. Furthermore, no α-renaming is required since identifier naming convention in Haskell subsumes that of ABS.

Primitive Types

The Standard ABS defines the Int and Rat arbitrary-precision number primitives. For execution performance reasons, the HABS implementation restricts those two to fixed-precision, native-architecture counterparts, e.g. Data.**Int**.Int64 and Data.**Ratio**.**Ratio** Int64 for 64bit computer architectures. An integer computation that "overflows" will not trigger an exception in Haskell. However, supporting arbitrary-precision numbers (i.e. **Integer** and **Rational** in Haskell) would not require a major refactoring of the HABS compiler.

The String primitive of ABS is translated to the **type String** = **[Char]** in Haskell, which as the definition suggests is implemented as a single linked-list of unicode characters. There exist faster alternatives for Haskell (e.g. the `bytestring` and `text` libraries), but for the moment this does not add much since usually ABS models do not do heavy string manipulations; this may change in the future.

Futures of ABS (Fut<A>) are represented in Haskell by the Control.Concurrent.MVar, which is a mutable variable living on the global heap, which contains some value A. Unlike the usual mutable variables of Haskell (IORef), MVars are concurrent datastructures which support for synchronization and fairness. The use of MVars for ABS concurrency is detailed more in the section 3.5 about HABS' runtime execution.

Algebraic Datatypes

Algebraic Datatypes of ABS correspond one-to-one to Haskell's simple algebraic datatypes; both are immutable datastructures, the difference being only syntactic, e.g. type variables in ABS are upper-case whereas in Haskell are lower-case, etc.. In fact, the Haskell type system can define more expressive datatypes than those of ABS, e.g. generalized algebraic datatypes (GADTs), existential quantification and datatype contexts.

ADT accessors of ABS are translated to Haskell (partial) pure functions. For example:

```
data User = Human (String name)
          | Bot(String name, Int version );
```

The above ABS code will result to the following Haskell ADT and two function "accessors":

```
data User = Human String
          | Bot String Int;

name :: User -> String
name (Human s) = s
name (Bot s _) = s

version :: User -> Int
version (Bot _ i) = i
```

Type Synonyms

Unlike most other constructs, type synonyms are a "preprocessing" construct and do do not carry any runtime costs, i.e. they are only used during typechecking phase and are omitted at code generation phase which strips off any types. As such, the ABS type synonyms are translated by the HABS transcompiler to the Haskell equivalent ones, which will be typechecked and discarded by the GHC compiler. Haskell by-default supports parametric type synonyms. We also rely on a new feature of Haskell called `PartialTypeSignatures` to support (partial) type inference in HABS.

3.3.3 Stateful code

As discussed in the Section 1.1, ABS has been designed to be familiar to programmers using the main-stream object-oriented style of programming. The

question arises on how we can implement the high-level, familiar concepts of object-oriented programming in Haskell. It is less straightforward to translate the ABS language's local variables and object fields to Haskell, compared to for example translating to a classic, imperative language: Haskell is a *purely* functional language and as such there exists no builtin notion of (implicit) side-effects. This, however, does not mean that Haskell cannot represent stateful code at all; in fact, stateful computation in Haskell can be (a) more expressive and (b) safer than most imperative languages, because of (a) the option of constructing multiple monads each having different effects and combine them (by monad transfomers) under a larger monad and (b) the clear separation at the type-level of pure and side-effectful code, thanks to the *monad* abstraction. Monads are a well-studied concept in the Category Theory of mathematics; here, for practical purposes, we can think of a monad as a typed computation that has an *explicit set of effects* and provides two operations around those effects: sequencing effects (`;` in imperative languages, `>>=`, `>>` in Haskell) and "lifting" pure expressions to look as they are effectful (**return** in Haskell). Since such monadic computations are statically-typed, the type-system does not allow us to include monadic code inside pure code — the opposite is safe though and is done through **return**. Even further, there exist different monads (offering perhaps different sets of effects) and the type system, again, will not permit any implicit intermix of monadic code belonging to different monads; any such conversion of monads (and their effects) have to be explicit.

One of the most common monads provided by Haskell is the so-called State monad. This monad allows the underlying computation to keep track of some state (represented as data e.g. an ADT), as well as access it or modify it during the whole computation. This State monad can be implemented in Haskell itself as the function with `type State s a = s -> (a, s)` where is s is the state data and a is the result of the whole computation.

The other most well-known monad in Haskell is the **IO** monad, which as the name suggest is used for input & output to the screen, file, network, etc.. This monad can be considered as a particular instance of the described State monad, and given as the type synonym: `type IO a = State RealWorld a` where $RealWorld$ is the current state of the whole natural world and a is the result type of the IO computation. However, for "practicality reasons" the RealWorld datatype is not representable in Haskell and as such is a "magical", abstract datatype. Similarly, the actual implementation of IO does not use the purely-functional *State* monad but instead the primitive *State#* monad, which is implemented in a low-level C library.

For implementing (local) mutable variables and objects of ABS, we decided not to use the pure *State* monad, but instead the IO monad for two reasons:

a) it makes certain imperative constructs easier to define (e.g. while) and b) it allows implementing exception handling for the ABS actor system; exceptions between threads in Haskell are asynchronous and (generally) primitive, so they exist only in the IO monad. For HABS, the ABS main block and all the bodies of methods (which are sequences of statements) become stateful (monadic) code. As mentioned earlier, Haskell disallows the inclusion of monadic code inside pure code at the type-level; consequently, the ABS-translated code is also guaranteed (by the type-system) to not mix side-effectful ABS object-oriented code inside purely-functional ABS code.

Mutable variables in Haskell

One particular effect that the IO monad provides, is access to the global memory heap of the program. This is realized by $IORef$ which is an abstract reference to a memory location inside the heap [2] We can allocate a new reference by calling $newIORef :: a- > IO(IORefa)$, which given any data typed by a will store them in the heap and return a reference to them. As such, the $IORef$ acts as a container of data in the heap where the data can be read back (dereferenced) by calling $readIORef :: IORefa- > IOa$ or changed by calling $writeIORef :: IORefa- > a- > IO()$. The data inside the $IORef$ will remain "alive" (not garbage-collected) at least as long as the $IORef$ remains alive. An $IORef$ reference can be passed around, composed, and stored inside other $IORef$s as usual data.

We give an example of an ABS snippet accessing mutable variables, which is translated to Haskell through the HABS compiler:

```
{
 Int x = 3;
 Int y = 4;

 x = y + 1;
}
```

```
main = do
 x :: IORef Int <- newIORef 3
 y :: IORef Int <- newIORef 4

 writeIORef x =<< ((+) <$!> readIORef y <*> return 1))
```

[2] The $IORef$ should not be confused with the C pointer, which is a fixed memory address, since $IORef$'s may transparently change their underlying memory address during a garbage collection phase.

Since $IORef$s live in the (shared-memory) global heap, they are susceptible to race conditions. However, for the case of HABS, we can assume that no such race conditions of ABS mutable variables will happen, as long as the HABS to Haskell compiler does not contain an implementation bug on the described translation.

Note that, although, Haskell does keep a call stack (like lower-level languages), any data from local variables of the stack frames are not stored directly inside the stack datastructure, but simply referenced from the stack to a different heap location that contains the actual data.

3.3.4 Object encoding

An object is a specific *instance* of a class and thus holds a separate "copy" of all the non-static members (fields or methods) of its class. Since objects are usually long-lived and/or large (contain a lot of fields/methods), they are (most commonly) stored on the heap (instead of the stack). An object will thus usually be a contiguous memory chunk containing (among other information) its fields and a virtual table of methods for dynamic-dispatching.

Similarly for HABS, an object (instance) is represented as a Haskell record of its fields. A Haskell record is the same as an immutable algebraic datatype of ABS where each field name acts as an accessor, e.g. in Haskell code:

```
data ClassContents = ClassContents(field1Name :: Field1Type,
                                   field2Name :: Field2Type,
                                   ...);
```

Thus ABS classes become algebraic datatypes (ADTs) acting as record types (containers) of their fields, and objects become merely values (instances) of such record types. Since record values in Haskell are immutable and we perhaps need to mutate an object's fields at runtime, we allocate a mutable reference ($IORef$) to hold the object's contents (record value). The type of an *object reference* is given in HABS implementation as:

```
data ObjRef contents = ObjRef (IORef contents) Cog
```

where contents is a type variable for the container type (in the example would be the ClassContents datatype) and Cog is a reference to the object's group — you can find more about the cog's representation in section 3.5 about HABS' runtime execution. Thus, the statements new Class() and new local Class() in ABS corresponds to the creation of a new $ObjRef$ and allocation of its $IORef$ contents, plus the execution of the init-block of the $Class$.

An alternative implementation would be to have for each object an immutable record of mutable references, e.g. in ABS syntax:

```
data ClassContents = ClassContents(field1Name :: IORef Field1Type,
                                   field2Name :: IORef Field2Type,
                                    ...);
```

which although leads to faster field accesses (and finer-grained await-on-boolean implementation), it has the theoretical downside of putting more garbage collection pressure, since the garbage collector will have to scan more mutable references in the global heap.

Note that, contrary to a canonical implementation of objects inside the heap, the Haskell object-reference type does not carry a virtual table of methods. This is instead stored separately on a wrapper datum which carries the current interface type of the object — see the section 3.3.5 on the runtime representation of interfaces and methods in HABS.

3.3.5 Interfaces, Classes and Methods

An ABS interface declaration is represented in the translated Haskell code by a *typeclass*. We give such a translated example from ABS code taken from section 2.4.2:

```
class InterfName1' a where
        method1 :: List Int -> ObjRef a -> IO Int
class InterfName1' a => InterfName2' a where
        method2 :: Int -> ObjRef a -> IO Bool
```

Typeclasses are a Java interface-like feature that first appeared in Haskell, which when combined with the parametric polymorphism, leads to ad-hoc polymorphism more powerful than commonly found in mainstream languages (Java, C++). Methods are monadic actions: their Haskell type is of the form Arg1Type -> Arg2Type -> ObjRef a -> **IO** ResultType, where the reference to the object callee this is passed as the last argument to the method (ObjRef a in the method's type).

ABS classes become **instance**s to the Haskell typeclasses (ABS interfaces). A Haskell typeclass instance provides an implementation for the functions (methods in our case) described inside the typeclass (ABS interface). An example of a particular ABS class is given:

```
class C implements InterfName1 {
  Int method1(List<Int> y) {
```

```
    return 3;
  }
}
```

which is translated to Haskell by the HABS compiler as:

```
instance InterfName1' C where
        method1 y this = do
            return 3 -- translated (sub)-expression
```

Unlike other statically-typed, object-oriented languages which perform type erasure at compile-time, an object reference in HABS will be wrapped with its current interface (which subsequently holds the virtual table of methods at runtime):

```
data InterfName1 = forall a . InterfName1' a => InterfName1 (ObjRef a)
data InterfName2 = forall a . InterfName2' a => InterfName2 (ObjRef a)
```

In Haskell this technique is called existential quantification (despite the $\forall$ symbol), which acts as an existential wrapper over an ABS object reference. This wrapper attaches (at runtime) the "name" of the current interface type (nominal typing) of an object reference as well as a link to a virtual table of method implementations for dynamic dispatching of (synchronous & asynchronous) method calls. It becomes obvious that this technique incurs an extra performance cost at runtime for holding the current interface wrapper as live data on the heap, instead of having the types erased after compilation. This performance cost becomes more apparent when implementing the (covariant) subtyping of HABS inside the Haskell language which is discussed in section 3.4.1.

To conclude the overall translation of ABS to Haskell, the module system is one-to-one translated to its very much alike Haskell equivalent; the ABS standard library exists in two versions: 1) the "slow" version implemented in ABS itself and (re)compiled to Haskell on each execution of the HABS compiler 2) a "fast" version where most of the ABS standard library is implemented directly in Haskell using optimized Haskell-provided datastructures (Set and Map) and imported to the translated Haskell code as a fixed Haskell module. The fast version supports better integration with the foreign language interface of Haskell, since certain standard datatypes will correspond to Haskell equivalent ones (e.g. List<A> of ABS becomes [a] in Haskell) and thus any foreign Haskell code which uses the latter can be safely imported to ABS. The downside of the fast version is that it is non-portable (to other backends) and susceptible to any changes to the overall ABS standard library; such changes

would require manually modifications to the fast version of the HABS standard library. Finally, since delta meta-programming (Section 2.6) is similar to preprocessing, it happens early on in the compiler frontend phase of any ABS code and thus all ABS backends will compile only the macro-expanded ABS code, free from any deltas.

3.4 Typing ABS

Standard ABS, as shown in section 2.4, is statically-typed with a type system that offers both parametric polymorphism and nominal subtype polymorphism. Our implementation of HABS focuses mostly on correct (i.e. faithful to ABS semantics) source-to-source compilation of ABS into Haskell; for this reason and the reason that the type-systems of ABS and Haskell have commonalities, a large part of type-checking is left to be performed by the Haskell typechecker itself. Specifically, we rely on the Haskell typechecker for both parametric polymorphism and partial type inference (for non-interface types): a recent version of GHC's typechecker (version $\geq$ 8.0.1) is needed with support for both parametric polymorphism and partial type inference with the `PartialTypeSignatures` language extension. The translation of such HABS types to Haskell equivalent is straightforward and thus omitted from this thesis. In the rest of this section on typing HABS, we only discuss the rest of the ABS type-system, i.e. subtyping and foreign-language interface, which has to be typechecked by the HABS compiler during the translation and simply cannot be left to a Haskell typechecker, since the Haskell language does not support any form of subtyping out-of-the-box.

The upside of not performing full type-checking for HABS, and instead partly relying on the "target" typechecker, is that we benefit from the proven GHC type-checking implementation; however, the main drawback is that the HABS type errors are usually incomprehensible, because they reflect the Haskell translated code and not the original ABS code — a common problem in source-to-source compilation and embedded domain specific languages, in general. Indeed, 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 our typechecking method; in other words, the Haskell typechecker cannot be fully aware of all the ABS language constructs. Nevertheless, any HABS-generated program will be ABS-type safe, in the sense that all type errors are caught at compile time and no type-error escapes to runtime.

3.4.1 Subtyping

Haskell's type system does not support any form of subtyping (structural or nominal) out of the box; for this reason, we cannot completely rely on Haskell's typechecker. Instead, we add support for nominal subtyping of ABS directly to the HABS compiler itself. The Standard ABS language specification defines *implicit* upcasting of interfaces, with no mentions of any (safe) downcasting. The HABS compiler implements such upcasting by wrapping identifiers (local variables or fields) that are typed by interface, with an upcasting function (named `up`). This function is overloaded by a `Sub` typeclass, declared in Haskell as:

```
class Sub sub sup where
    up :: sub → sup
```

For each subtype relation (of interfaces), the HABS transcompiler will accordingly generate *boilerplate* instances of the above upcasting typeclass. Consider for example the three ABS interfaces:

```
interface I1 {}
interface I2 extends I1 {}
interface I3 extends I1 {}
```

The HABS compiler will generate, other than the particular interfaces and its interface wrappers shown in section 3.3.5, specific Haskell code for their upcasting-relation instances as:

```
instance Sub I1 I1 where
        up x = x
instance Sub I2 I2 where
        up x = x
instance Sub I3 I3 where
        up x = x
instance Sub I2 I1 where
        up (I2 a) = I1 a
instance Sub I3 I1 where
        up (I3 a) = I1 a
```

Note that the `null` ABS construct can be typed by any interface type; however, there is no "root" interface type in the ABS interface hierarchy (e.g. compared to Java's `Object` class). An example of ABS code that relies on upcasting is the following trivial function:

```
def I2 f(I1 obj) = obj;
```

which translates using the HABS compiler to the Haskell code:

```
f :: I1 -> I2
f obj = up obj
```

This particular method of wrapping identifiers with the up function works fine for simple cases of subtyping, as in the above example. The method's problem appears on ABS code that requires *implicit* upcasting, e.g.:

```
// the builtin equality function in ABS is defined as
def Bool (==)<A>(A l, A r) = <internal_implementation>;

{
 I2 obj2;
 I3 obj3;
 Bool b = obj2 == obj3; // implicit upcasting to least-common super interface
}
```

Following the simple method (of just wrapping each identifier in the Haskell generated code with a call to up), leads to type ambiguity problems by the subsequent Haskell typechecking, since its typechecker cannot compute a common interface to upcast the two objects to:

```
up (obj2 :: I2) == up (obj3 :: I3) -- TYPE ERROR: Haskell ambiguous type
```

To fix this, the HABS compiler keeps track of the complete nominal subtype hierarchy of the ABS program under compilation and computes the least-common super-interface type — if it exists, otherwise signals a type-error. The least common super-interface, whenever needed, is added by HABS to the generated Haskell code in the form of extra type signatures that remove any Haskell type ambiguities. The example before will be annotated by HABS with type signatures of $I1$ least-common super interface, which will be accepted later by the Haskell typechecker:

```
(up obj2 :: I2) :: I1 == (up obj3 :: I3) :: I1
```

This approach using extra type signatures solves the problem of implicit upcasting in ABS. However, yet another problem persists: that of variance.

Adding least common interfaces as extra type signatures solves the problem of implicit upcasting for HABS, but it is not enough to express the full type-system of Standard ABS in terms of Haskell, specifically because of variance support. As discussed in section 2.4.3, the specification of Standard ABS leaves the (default) type variance undefined; however, given its current language

standard, it is safe to assume that only *covariance* is needed for ABS types (mostly datatypes combined with interface types). This happens to be the case for other ABS compilers (Maude-ABS, Erlang-ABS) where they offer such support for covariance. In the future, if the ABS language standard is augmented with first-class functions and/or polymorphic methods, other types of variance (contravariance, invariance) may be needed. Coming back to HABS, consider an ABS snippet which exhibits covariance:

```
{
  List<I2> l2 = list[obj2];
  List<I1> l1 = Cons(obj3, l1);
  List<I1> l1 = l2;
}
```

In the second line, and according to our translation scheme, the `obj3` would be correctly wrapped with the `up` function (concretely: `(up obj3 :: I1)`). Unfortunately, in the third line we cannot wrap as well the identifier `I2` with `up`, since the upcast function operates on ground interface types (i.e. `up :: sub -> sup`) and not on (arbitrary) algebraic datatypes mixed with interface types. In other words our `up` function is not enough and we would hypothetically like to have an extra `upList :: List<sub> -> List<sup>`. We could instead utilize a similar function already existing in Haskell called `fmap`, (for functor-map) to map `up` over each "substructure" of list; our translated code (simplified for sake of clarity) would be well-typed in Haskell as:

```
do
  let l2 = [obj2]
  let l1 = (up obj3 :: I1) : l1
  let l1 = fmap up l2 :: [I1]
```

This solution does work for simple ABS cases of covariance for ABS single-arity functor data types (e.g. `List<A>`, `Maybe<A>`) but becomes problematic for arbitrary-arity functors, for example bifunctors (`Either<A,B>`), trifunctors (`Triple<A,B,C>`) and so on and so forth, since no "generic" *fmap* function over any arity exists. Instead, we use the *genifunctors* library `https://hackage.haskell.org/package/genifunctors` which in turn makes use of Template Haskell (macro meta-programming) to generate a separate fmap-like function specific for each ABS datatype defined (builtin or user-defined). Consider the ABS example:

```
{
  Either<Bool,I1> e = Right(obj2);
  Triple<I1,Unit,I1> t = Triple(obj2, Unit, obj3);
```

```
}
```

HABS generates the following Haskell code:

```
do
  let e = fmapEither id up (Right obj2) :: Either Bool I1
  let t = fmapTriple up id up ( Triple  obj2 Unit obj3) :: (I1 , Unit , I1 )

fmapEither :: (a ->a1) -> (b -> b1) -> Either a b-> Either a1 b1
fmapEither f g x = case x of
    Left x1 -> Left (f x1)
    Right x1 -> Right (g x1)

fmapTriple :: (a ->a1) -> (b -> b1) -> (c -> c1) -> (a,b,c) -> (a1,b1,c1)
fmapTriple f g h ~(a,b,c) = (f a , g b , h c)
```

where fmapEither and fmapTriple are the simplified, macro-expanded boilerplate code generated by the *genifunctors* library.

This subtyping technique of HABS discussed up to here is regarded in the object-oriented field as *coercive subtyping*: the objects carry at runtime their currently-typed interfaces (in the case of HABS as interface existential wrappers) and an accompanying generic function up will *coerce* (in the sense of change the data structure's representation) at runtime any interface type to a super interface type (and its covariants). The other most-used technique in mainstream object-oriented implementations (Java, OCaml) is called *inclusive subtyping*, where most types can be erased after compile-time since the object memory layout at runtime is compatible with all of its super-interfaces; in other words, there is no need for an upcasting function to be applied at runtime so as to perform any object layout changes (coercion). The largest drawback of coercive subtyping is that there is runtime performance costs of performing the actual coercion, i.e. changing the objects' structure itself or transforming a data structure (fmap) that includes the object(s). Theoretically, there is a minor benefit of coercive over inclusive subtyping, in the sense that, during a runtime upcasting operation an object can garbage-collect a portion of its attributes (e.g. fields) which are unnecessary for super-interfaced methods (assuming downcasting is not allowed by the language). This is exploited in the case of HABS and the Haskell/GHC garbage-collector.

Concerning Haskell and subtyping in general, the approach in [Kiselyov et al., 2004] and its further development in [Kiselyov and Laemmel, 2005] employ heterogeneous lists and type-level programming to extend Haskell with even more object-oriented concepts than needed for the sake of translating ABS, e.g. class code inheritance,

multiple inheritance, contravariance, depth subtyping. A new and promising approach is to use the `Generic` metadata representation found in GHC version 8.0 to perhaps remove (some of) the boilerplate code-generation which relies on Template-Haskell and instead employ the Haskell's native datatype generic programming [Magalhães et al., 2010]. Yet both these two described approaches would still implement *coercive* subtyping for Haskell (and its HABS "embedding"). To the best of our knowledge, there is currently no published work that addresses inclusive subtyping for Haskell; this may be perhaps attributed to the current limitations of GHC's memory heap layout. In the worst case, inclusive subtyping for Haskell/GHC would require an extension of Haskell's type system with "first-class" support for subtyping.

3.5 Runtime execution

The translated Haskell code is linked against our custom concurrent runtime library, which is based on GHCs (Glasgow Haskell Compiler) own runtime system (RTS). This library adds the concurrency model of ABS to Haskell; more specifically, the high-level features of cooperative scheduling, awaiting on futures, and awaiting on booleans of ABS can now be used and intermixed with native Haskell code. Our runtime-as-a-library and its features can hypothetically be used completely outside of ABS and directly inside Haskell code; in addition to the automatic default object-encoding provided by the HABS compiler, the user can also manually choose an encoding and subtyping of their choice.

Each ABS Concurrent Object Group (COG) is represented in our runtime by a separate Haskell lightweight thread (also known as green thread or userspace thread). Such threads differ from the system threads commonly found in other languages (e.g. Java, C), since they carry a smaller memory footprint and are managed (scheduled) not by the underlying operating system (OS), but directly from the language's runtime system. Since Haskell threads are very lightweight, a HABS execution could contain "millions" of COGs inside a single machine, without running out of memory.

GHC's runtime system goes a step further by offering an *M:N* threading model: the RTS manages M lightweight Haskell threads and schedules them for execution over N system threads, all the while automatically load-balancing them (through a preemptive scheduling scheme). This hybrid threading model of GHC also benefits from the Symmetric Multi-Processing (SMP) support of the operating system, for the *parallel* execution of Haskell threads by multi-core CPUs.

Each COG-thread retains an ABS process-queue (similar to an actor's mailbox) that holds processes to be executed; a new ABS *process* is created and put at the end of the queue upon an asynchronous method call. Every COG-thread listens to its own process queue for new or re-activated processes and executes one at a time up to their next release point (await or return).

Processes are implemented as coroutines (which are themselves implemented as first-class continuations) and not as threads, which allows us to store them inside the COG's 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) and when invoked, will replace the current state of the program with the continuation's saved state. Continuations are *initially* created by asynchronous method calls: an asynchronous method activation pushes a new continuation to the end of the callee's process queue. In other words, during such an asynchronous method call, a caller creates a new process by applying the corresponding function to its arguments and stores its body (function closure) at the end of the callee's COG queue.

The evaluation of the suspend ABS statement captures the current continuation of the running process and stores it in the end of its COG's process-queue (for later resumption). The program is at a release point and so the execution then jumps to the *main loop* of the COG, which contains a blocking read from the head of the process-queue for selecting another process to resume. This suspension-resumption procedure is the simplest form of cooperative multitasking for HABS (and the ABS language).

Processes awaiting on boolean-conditions (e.g. await booleanExp;) are continuations which will be captured and resumed only when their condition is met. The naive approach to implement is to regard boolean awaiting as a form of syntactic sugar of a while loop that suspends, e.g.:

```
Unit m() {
  before ...;
  await (this.x>this.y+1);
  after ...;
}

// desugared as
Unit m() {
  before ...;
  while !(this.x>this.y+1) {
    suspend;
  }
  after ...;
```

```
}
```

However, such implementation leads to *busy-await polling* (and consequently waste of CPU cycles) since we resume the process even if its conditions are guaranteed to not have been met yet. Instead, we use a refined approach where we store inside each COG thread, besides a process-queue, a "SleepTable" which is an association list of boolean actions to continuations, hence the type **type** SleepTable = [(**IO Bool**, ABS' ())]. We also modify its COG's main-loop to traverse the "SleepTable" at every release point and remove the first continuation that its associated action (**IO Bool**) evaluates to **True**; intuitively the action computes the current value of its ABS boolean expression. If such a continuation exists then the COG will immediately remove it from the SleepTable and resume it, otherwise the COG will fall back to block on reading from its process-queue (mailbox) as before. A new entry is inserted to the SleepTable upon a new boolean-await statement call; the table does not have to be updated when any field is modified, since field values are extracted from the latest object reference IORef, hence the monadic action **IO Bool**. A further refinement to this "testing" of boolean-awaiting continuations that we did experiment with, is to use a "monitor"-like implementation, where the "SleepTable" becomes instead an association of object field indices to continuations: the continuation will be tested only in the condition that at least one of its dependent fields — an ABS boolean expression can only change because of *this.field* modifications — has been modified since the previous time of its testing; in other words retrying only those continuations that have part of its condition modified (by mutating fields) since the last release point.

Continuing on, awaiting on futures also avoids similar busy-wait polling by making use of the asynchronous I/O event notification system of the underlying Operating System (e.g. epoll on Linux, kqueue on *BSD), which the GHC runtime system is interfacing with. When a process decides to await on a future (by calling await f?;), a new separate lightweight thread is created with its captured continuation placed inside. This newly-created thread will block until its associated future has been completed; upon "unblocking", this thread will send its enclosing continuation back to the end of the original COG's process queue (again for later resumption) and exit. The runtime system guarantees that such extra threads will not be re-scheduled (consume any resources) at least until their associated futures are completed.

Each future (Fut<A>) is implemented in HABS as a concurrent datastructure residing in the memory heap. Such a datastructure will either be empty (not completed yet) or full containing the result. Any number of threads may block until the datastructure is full; one thread will write back the result, effec-

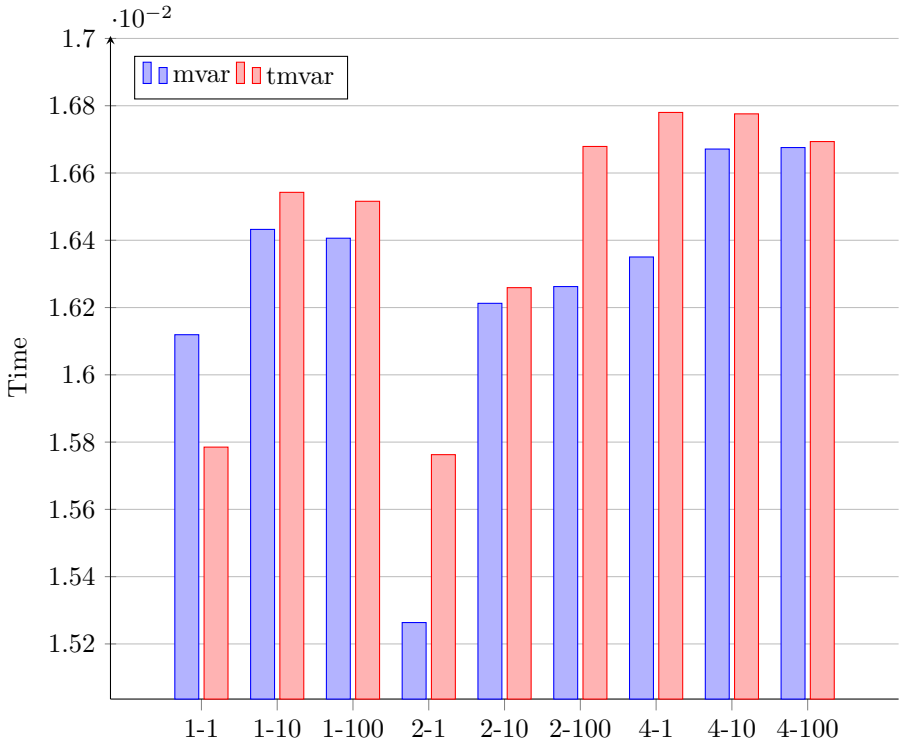


Figure 3.1: Implementing futures using MVar or TMVar on varying scenarios (workers-listeners).

tively waking up all blocked threads. In Haskell and GHC, such a concurrent datastructure can be realized by the Standard Library's MVar (standing for mutable variable) or TMVar (software-transactional-memory MVar). The difference between the two is that MVar guarantees *fairness*, i.e. blocked threads will be woken up in the order they arrived (FIFO). Since ABS semantics do not impose any fairness restrictions on how processes should be woken up when a future is completed, we decided to benchmark both implementations. On a system of 2-cores, 4 hyperthreads, the MVar datastructure seems to be generally slightly faster than its TMVar counterpart — the results are shown in figure 3.1).

Finally, although the HABS semantics leave the ordering of processes inside each COG unspecified, we decided to implement a "mailbox" of processes

as a FIFO queue. This choice is motivated by the fact that a FIFO queue preserves the "local" ordering of asynchronous method calls; for example, executing `o!m1();o!m2();` is guaranteed to not pick for execution the `m2` call before the `m1`, something which is usually expected by users (of imperative programming). Thus, for this HABS parallel runtime, the mailbox is represented by a concurrent datastructure residing in the heap; "sending" an asynchronous method call "writes" the continuation data to the end of the queue. Many different concurrent FIFO queue implementations exist for Haskell and GHC e.g. Chan, UnagiChan, TChan, TQueue; we benchmarked some of them and decided to go with a `TQueue` implementation, modified for the continuation monad, which as the results show (figure 3.2) is overall fast and almost as fast as the plain `TQueue` implementation (with no cooperative multitasking approach). Note that the process queue is concurrently modifiable which means that the COG thread can continue "popping" processes from the head of the queue and executing them all the while. In parallel, object-callers are placing new asynchronous method calls and processes awaiting on futures are resolved.

3.6 Comparison to other ABS Backends

Besides HABS, there have been other backend implementations for ABS, with the most complete of those (as of 2017) being:

Maude-ABS The Maude-ABS backend is used for prototyping and testing the ABS semantics in the Maude term-rewriting system.

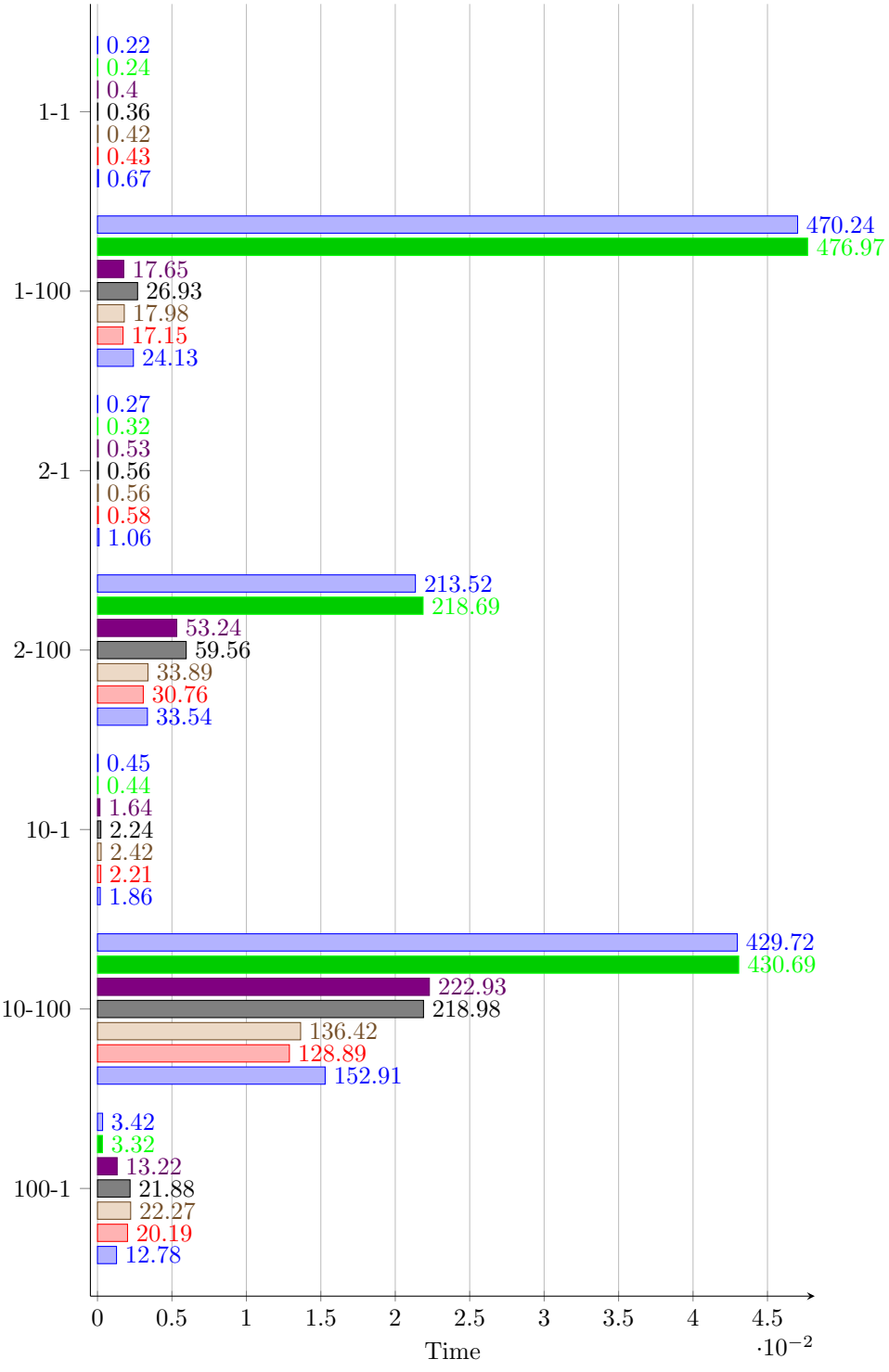
Java-ABS The Java-ABS backend was the first backend specifically developed to implement the Concurrent Object Groups (COGs) and has been superseded by the Erlang-ABS backend.

Erlang-ABS This backend is the currently most-used and maintained backend and is written in the Erlang programming language. It provides a reference implementation for the simulation of ABS models.

Java8-ABS The Java8-ABS backend makes use of recent Java technologies (lambda abstractions, thread-pools) to deliver a better performance for ABS executions than the above Java-ABS backend.

3.6.1 Comparing language support and features

The Maude-ABS backend is the backend of choice for designing, testing and experimenting with new language features of ABS; in this respect, the Maude-


1-1
0.22
0.24
0.4
0.36
0.42
0.43
0.67
1-100
470.24
476.97
17.65
26.93
17.98
17.15
24.13
2-1
0.27
0.32
0.53
0.56
0.56
0.58
1.06
2-100
213.52
218.69
53.24
59.56
33.89
30.76
33.54
10-1
0.45
0.44
1.64
2.24
2.42
2.21
1.86
10-100
429.72
430.69
222.93
218.98
136.42
128.89
152.91
100-1
3.42
3.32
13.22
21.88
22.27
20.19
12.78
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
Time
$\cdot 10^{-2}$

ABS backend is likely the most feature rich of all ABS backends. Besides the language differences discussed in section 3.1, an extra feature of the HABS implementation currently missing from the other backends is the support for runtime deadlock detection, i.e. knowing that (some) awaiting ABS processes cannot continue because of mutual dependencies. This is achieved thanks to the Haskell GHC's garbage collector detection. On the other hand, there do exist ABS *static-analysis* tools that search for possible program deadlocks [Albert et al., 2014a, Giachino et al., 2016a].

Most ABS backends and tools are integrated with the Envisage Collaboratory [Doménech et al., 2017] `http://abs-models.org/laboratory`, a web-based IDE for interactive experimenting with the ABS language and toolsuite without requiring any program installations: instead, the ABS backends and tools are installed on a web server and the client (user) just remotely interacts with them. The HABS backend also happens to be supported by the Envisage collaboratory; for the future, we are considering using *ghcjs* `https://github.com/ghcjs/ghcjs`, a Javascript backend for GHC, to compile ABS user-code on the server-side through HABS and ghcjs directly to Javascript, and execute it only at the client side: in this way we benefit by not executing unsafe user code on the server side (no need for sandboxing), and relieving the collaboratory server system from excessive computing resources.

3.6.2 Comparing runtime implementations

As opposed to some other backends (Erlang-ABS, Java-ABS), the Haskell backend does not treat active ABS processes as individual system threads, but instead as data (closures) that are stored in the queue of the concurrent object, which leads to a smaller memory footprint. This "data-oriented" implementation preserves local message ordering of method activations, although the ABS language specification leaves this unspecified.

Maude's term rewriting approach allows easy experimentation with ABS semantics and model-checking of ABS programs. Since it can explore all execution paths of an ABS model, it can replicate the local message ordering of HABS by following strictly specific execution paths. The largest drawback of the Maude-ABS backend is its slow execution speed (as later shown in section 3.6.3) which makes it unsuitable for programming. The Maude-ABS backend also has only very limited I/O capabilities, which deems for example the new HTTP-API extension for ABS difficult to implement.

The Erlang-ABS backend relies on the Erlang runtime to implement actor-style concurrency for ABS. This backend offers simulation of ABS models based on timed automata, and is discussed in Chapter 4. In contrast to HABS, the

processes in the Erlang-ABS backend are not continuations (data) stored in a COG's queue, but alive Erlang processes (Erlang's version of lightweight, green threads) living on the heap. The processes of each COG are competing with each other to acquire a token: acquiring a token means that the process will try to resume its execution; releasing the token means that the process stumbled upon an execution of suspend or await. This process-based implementation of a COG's "mailbox" cannot guarantee the local message ordering as is the case with HABS.

The Java-ABS backend is the first "real-world" backend designed with performance in mind; the backend is however currently not maintained. The backend follows the data-based approach of continuations which is also employed by HABS, but the difference lies in implementation, since such continuations are not natively implemented but reified in the Java language itself. Since Java lacks native support for first-class continuations — it lacks tail-call optimization and until Java version 8 also lacked closures — the support for continuations is added in an interpreted-like fashion. The generated by the backend Java code manages its own stack frames, above those of the JVM.

The Java8-ABS backend does not follow such an interpreted approach, but similar to Akka, employs a fixed thread-pool where COGs get a chance to execute on. Depending on the ABS programs involved, this may lead to *process starvation* where a number of COGs occupy the threads and do not release their resources. HABS, on the other hand, does not suffer from such *process starvation*, since the number of (lightweight) threads (COGs) is not fixed and can grow indefinitely up to memory exhaustion.

3.6.3 Benchmarking the ABS backends

The improved execution performance is the main advantage that come with the ABS-to-Haskell backend, as can be witnessed by the benchmarks and experimental results in this section. The concurrency/threading model of Haskell proved to be well-suited for ABS' cooperative multitasking.

An important feature of the HABS backend presented in this section is its good performance compared to the rest of backends for the ABS language. To show this, we developed a series of sequential and parallel programs that try to cover all features of the ABS language and we executed them using the ABS backends: the HABS backend, Java-ABS and Java8-ABS backends, the Erlang-ABS backend and the Maude-ABS backend. The results appear in Table 3.2, where times are in seconds, memory usage in KB and a hyphen (-) means that the program got stuck. These synthetic ABS benchmarks programs can be found at `https://github.com/abstools/abs-bench`). The

benchmark results indicate that the HABS backend is the fastest both in terms of elapsed time and memory residency. Specifically, the HABS backend is on average **13x** faster while taking up **15x** less memory than the Java8-ABS backend; this may be attributed to the fact that the Java8-ABS backend relies on Java's heavyweight threads. Two other downsides of the Java8-ABS 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 Java8-ABS backend they hang, because the employed threading model (static threadpool) limits how many "processor" units (COGs) can run concurrently. The Java-ABS backend is slower than the newer Java8-ABS backend, and consequently slower than the HABS backend (**256x** more time and **84x** more memory); the reason may be attributed to factors affecting also the Java-ABS backend and also the fact that the Java-ABS backend uses busy-waiting when monitoring active objects for their *await* conditions. As Table 3.5 shows, the Erlang-ABS backend got stuck in 3 of the 10 benchmark programs, so the comparison between the Erlang-ABS and HABS backend should be considered less reliable. Nevertheless, the Erlang-ABS backend takes **596x** more time and **17x** more memory than the HABS 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-ABS backend is extremely slow compared to all other backends since it is an interpreter, but surprisingly consumes comparable memory to HABS (**9x** more memory than HABS), and even in some cases less memory than the other 3 backends: Java-, Java8- and Erlang-ABS.

```
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 (build 1.8.0_60-b24) (build 25.60-b23, mixed mode)
Erlang 18 [64-bit] [smp:4:4] [async-threads:10] [hipe] [kernel-poll:false]
Maude 2.6 built: Dec 9 2010 18:28:39
```

The benchmarks of the ABS backends shown here can be better regarded as micro-benchmarks: benchmarks that stress-test the way that certain ABS features (concurrency,parallelism,object-creation) by the backends, but do not represent a real-world scenario of computational load. To this end, we constructed an ABS model that implements in a very high-level cache-coherence protocol, commonly found in everyday modern multi-core central processing units (CPUs). The ABS model is derived from that of a formally-verified model defined in Maude [Bijo et al., 2016].

Program	**Time(s)**	**Cpu(%)**	**Mem(KB)**	**÷habs-time**	**÷habs-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

Table 3.2: HABS

The results of this cache-protocol benchmark is shown in table 3.7, where Model Size refers to the number of processor cores of the *simulated* CPU, given its particular cache configuration. The performance results show that HABS backend was for this specific benchmark around 40 to 70 times faster than the Erlang-ABS implementation. The experimental setup was a 2-core, 4 hyperthreads Intel m-y10c system, running Windows 10 x64, Erlang-ABS v1.5.1, Erlang/OTP v20, HABS 6365791, Haskell GHC v8.0.1.

For the future, a larger and established set of real-world and micro benchmarks in the spirit of [Imam and Sarkar, 2014, Brandauer et al., 2015] would be greatly beneficial for the ABS ecosystem.

3.7 Formal verification of HABS

The overall contribution of this section is a formal, resource-consumption preserving translation of the concurrency subset of the ABS language into Haskell, given as an adaptation of the canonical HABS backend [Bezirgiannis and Boer, 2016]. This translation thus differs from the translation described above in section 3.3 ; this new, formal translation is detailed in section 3.7.3 together with a comparison between the two translations. We opted for the Haskell backend relying on the hypothesis that Haskell serves as a better middleground between execution performance and most importantly semantic correctness. The translation is based on

Program	**Time(s)**	**Cpu(%)**	**Mem(KB)**	**÷habs-time**	**÷habs-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

Table 3.3: Java8-ABS

Program	**Time(s)**	**Cpu(%)**	**Mem(KB)**	**÷habs-time**	**÷habs-mem**
BinarySearchTree	0.31	198	73340	31.00x	20.46x
FieldFutures	-	-	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

Table 3.4: Java-ABS

Program	**Time(s)**	**Cpu(%)**	**Mem(KB)**	**÷habs-time**	**÷habs-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

Table 3.5: Erlang-ABS

Program	**Time(s)**	**Cpu(%)**	**Mem(KB)**	**÷habs-time**	**÷habs-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

Table 3.6: Maude-ABS

Program Size	HABS Time(s)	Erlang-ABS Time(s)	÷habs-time
20	0.71	30.57	43.05x
50	1.47	67.49	45.91x
100	2.73	132.34	48.47x
200	5.01	355.69	70.99x

Table 3.7: HABS vs Erlang-ABS execution time for the cache-coherence protocol benchmark

compiling ABS methods into Haskell functions with *continuations*—similar transformations have been performed in the actor-based Erlang language w.r.t. rewriting systems [Palacios et al., 2015, Vidal, 2014] and rewriting logic [Noll, 2001], in the translation of ABS to Prolog [Albert et al., 2012] and a subset of ABS to Scala [Nakata and Saar, 2013]. However, what is unique in our translation and constitutes our main contribution, is that the translation is resource preserving as we prove in two steps:

- *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 ABS. However, since for efficiency reasons, the translation fixes a selection order between the objects and the processes within each object, we do not have a completeness result.
- *Resource-preservation.* As a corollary we have that the transformation preserves the resource consumption, i.e., the cost of the Haskell-translated program is the same as the original ABS program w.r.t. any *cost model* that assigns a cost to each ABS instruction, since both programs execute the same trace of ABS instructions. 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.

In Section 3.7.1 we specify the syntax of the source language and detail its operational semantics. Section 3.7.3 describes our target language and defines the compilation process. We present the correctness and resource preservation results in Section 3.7.4, as well as the intermediate semantics used in this process. In Section 3.7.6 we show that the runtime environment does not introduce any significant overhead when executing ABS instructions, and show that the upper bounds obtained by the cost analysis are sound. Complete proofs of the theoretical results can be found at Section 3.7.7.

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

Figure 3.3: Syntax of source language

3.7.1 Restricting to a subset of ABS

```
main() {
  node1 = new;
  node2 = new;
  f1 = node1!map(v1);
  f2 = node2!map(v2);
  await f1;
  await f2;
  r1 = f1.get;
  r2 = f2.get;
  r = reduce(r1,r2);
  return r; }

map(v) {
  ... }
reduce(v1,v2) {
  ... }
```

Listing 3.1: A simplified MapReduce task in ABS

Our language is based on ABS [Johnsen et al., 2010a], 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 support for *asynchronous* method calls; each call results in 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.

For this part, we simplify ABS to its subset that concerns the concurrent interaction of processes (inside and between objects), so as to focus solely on the more challenging part of proving correctness of the cooperative concurrency. In other words, the ABS language is stripped of its functional core, local variables, object groups [Schäfer and Poetzsch-Heffter, 2010] and types (we assume the input programs are well-typed w.r.t ABS type-system). The formal syntax of the statements S of the subset is shown in Fig. 3.3(a). Values in our subset are references (object or futures) and integer numbers; 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: attributes holding object references or integer values (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, an attribute x, an integer expression I (an integer value, addition, subtraction, etc.), 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. Again, note that, we assume expressions to be well-typed: integer expressions cannot contain futures or object references and boolean equality is between same-type values. 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. 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. 3.3(b) shows a basic version of a MapReduce task [Dean and Ghemawat, 2008] implemented using actors in ABS. For clarity the example uses only two *map* nodes and a single *reduce* computation performed in the controller node (the actor running main). First the controller creates two objects node1 and node2 (L2–L3), and invokes asynchronously `map` with different values v_1 and v_2 (L4–L5). In MapReduce, all `map` invocations must finish before executing the *reduce* phase: therefore, the await instructions in L6–L7 wait for the termination of the two calls to `map`, releasing the processor so that any other process in the same object of main can execute. Once they have finished, the get statements in L8-L9 obtain the results from the futures f1 and f2. Although get statements block the

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

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

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

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

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

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

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

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

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

Figure 3.4: Operational semantics: Local rules

object (in this case *main*) and all of its processes until the result is ready, this does not occur in our example because the preceding awaits assure the result is available. Finally, L10 contains a synchronous-method self call to reduce that combines the partial results from the *map* phase.

3.7.2 Operational Semantics

In order to describe the operational semantics of the language defined above we first introduce the following concepts and assumptions. The values considered in this section are in the *Int* set: integer constants and dynamically generated references to objects and futures. We denote by $\Sigma = \mathit{IVar} \to \mathit{Int}$ the set of assignments of values 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 variables by actual values (note that variables are introduced as method parameters and can only appear in E) and a future reference, represented by an integer, for storing the return value. By $S\tau$, where $\tau \in \mathit{LVar} \to \mathit{Int}$, we denote the instantiation obtained from S by replacing each 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) $h_1 : Int \to \Sigma$ and $h_2 : Int \to Int_\perp$, where $Int_\perp = Int \cup \{\perp\}$ ($\perp$ 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. 3.4 contains the relation that describes the local behavior of an object (omitting the standard rules for sequential composition, if and while statements). Note that the first closure of the list Q is the active process of the object, so 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 rule (Assign) modifies the heap storing the new value of variable x of object n. It uses the function $\texttt{getVal}(\Sigma, V)$ to evaluate an expression V involving integer constants and variables in the object's current state Σ. 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, thus avoiding starvation. 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 $d.m(l', \bar{v})$ 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 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 $\hat{\cdot}^s$ function, defined as follows (recall that `return` is the last statement of any method):

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

$$(\textsc{Message})\ \frac{\begin{array}{c}\langle n : Q_n, h\rangle \overset{d.m(l',\bar{v})}{\longrightarrow} \langle n : Q', h'\rangle \\ m(\bar{w}) \mapsto S_m \in D \qquad \tau = [\bar{w} \mapsto \bar{v}] \qquad S' = \widehat{(S_m\tau)}^{*}\end{array}}{\langle (n : Q_n) \cup (d : Q_d) \cup C, h\rangle \rightarrow \langle (n : Q') \cup (d : Q_d \cdot (S', l')) \cup C, h'\rangle}$$

Figure 3.5: Operational semantics: Global rules

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

Rule ($\textsc{Return}_A$) finishes an asynchronous method invocation (in this case the `return` keyword is marked with *, see rule (Message) in Fig. 3.5), 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. 3.5 shows the relation describing the global behavior of configurations. The (Internal) rule applies any of the rules in Fig. 3.4, 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. When needed, we decorate both local and global steps with object reference n and statement S executed, i.e., $\langle n : Q, h\rangle \rightarrow^{n}_{S} \langle n : Q', h'\rangle$ and $\langle C, h\rangle \rightarrow^{n}_{S} \langle C', h'\rangle$.

We remark that the operational semantics shown in Fig. 3.4 and 3.5 is very similar to the foundational ABS semantics presented in [Johnsen et al., 2010a], considering that every object is a *concurrent object group*. The main difference is the representation of configurations: in [Johnsen et al., 2010a] 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 [Johnsen et al., 2010a] left the scheduling policy unspecified.

3.7.3 Target Language

Our ABS subset is translated to Haskell with coroutines. A coroutine is a generalization of a subroutine: besides the usual entry-point/return-point of a procedure a coroutine can have other entry/exit points, at intermediate locations of the procedure's body. Simply put, a coroutine does not have to run to completion; the programmer can specify places where a coroutine can suspend and later resume exactly where it left off.

Coroutines can be implemented natively on top of programming languages that support first-class *continuations* (which subsequently require support for closures and tail-call optimization). A continuation with reference to a program's point of execution, is a datastructure that captures what the remaining of the program does (after the point). As an example, consider the Haskell program at Listing 3.3(a). The continuation of the call to (even 3) at L2 is λa$\rightarrow$ print a, assuming a is the result of call to even and the continuation is represented as a function. The continuation of (mod x 2) at L1 is the function λa$\rightarrow$ print (eq a 0) where x is bound by the even function and a is the result of (mod x 2). Abstracting over any program, an expression with type expr :: a has a continuation k with type k ::(a$\rightarrow$r) with a being the expression's result type and r the program's overall result type. To benefit from continuations (and thus coroutines), a program has to be transformed in the so-called *continuation-passing style* (CPS): a function definition of the program f :: args$\rightarrow$a is rewritten to take its current continuation as an extra last argument, as in f '':: args$\rightarrow$(a$\rightarrow$r)$\rightarrow$r. A function call is also rewritten to apply this extra argument with the actual continuation at point.

A CPS transformation can be applied to all functions of a program, as in the example of Listing 3.3(b), or (for efficiency reasons) to only the subset that relies on continuation support, e.g. only those functions that need to suspend/resume. For our case, ABS is translated to Haskell with CPS applied only to statements and methods, but not (sub)expressions. Continuations have the type k :: a$\rightarrow$Stm where Stm is a recursive datatype with each one of its constructors being a statement, and the recursive position being the statement's current continuation. Stm being the program's overall result type (Stm$\equiv$r), reveals the fact that the translation of ABS constructs a Haskell AST-like datatype "knitted" with CPS (Listing 3.4), which will only later be interpreted at runtime: capturing the continuation of an ABS process allows us to save the process' state (e.g. call stack) and rest of statements as data. For technical convenience, our statements and methods do not directly pass results among each other but only indirectly through the state (heap); thus, we can reduce our continuation type to k ::() $\rightarrow$Stm and further to the "nullary" function k :: Stm. Accordingly the CPS type of our methods (functions) and statements (constructors) becomes f '':: args$\rightarrow$Stm$\rightarrow$Stm. Worth to mention in Listing 3.4 is that the body of While statement and the two branch bodies of If can be thought of as functions with no args written also in CPS (thus type Stm $\rightarrow$ Stm) to "tie" each body's last statement to the continuation *after* executing the control structure.

A Method definition is a CPS function that takes as input a list [Ref] of the

method's parameters (passed by reference), the callee object named this, a *writeback* reference (`Maybe Ref`), and last its current continuation Stm. In case of synchronous call the callee method indirectly writes the Return value to the writeback reference of the heap 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. An object or future reference Ref is represented by an integer index 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). Values (V) in our language can be this-object attributes (A), parameters to the method (P), integer literals (I), and integer arithmetic on those values (Add, Sub...). The right-hand side (Rhs) of an assignment directly reflects that of the source language. Boolean expressions are only appearing as predicates to If and While and are inductively constructed by the datatype B, that represents reference and integer comparison.

```
even x = eq (mod x 2) 0
main = print (even 3)
```

Listing 3.2: Example program in direct style

```
mod' x y k = k (mod x y)
eq' x y k = k (eq x y)
even' x k = mod' x 2 (λ a → eq' a 0 k)
main = even' 3 (λ a → print a)
```

Listing 3.3: Example program translated to CPS

```
data Stm where -- (formatted in GADT syntax)
   Skip :: Stm → Stm
   Await :: Attr → Stm → Stm
   Assign :: Attr → Rhs → Stm → Stm
   If :: B → (Stm→Stm) → (Stm→Stm) → Stm → Stm
   While :: B → (Stm→Stm) → Stm → Stm
   Return :: Attr → Maybe Ref → Stm → Stm

data Rhs = Val V
         | New
         | Get Attr
         | Async Attr Method [Attr]
         | Sync Method [Attr]

type Ref = Int
type Attr = Int
```

$$
\begin{aligned}
{}^{s}[\![\texttt{skip}]\!]_{k,wb} &= \texttt{Skip}\ k \\
{}^{s}[\![\texttt{await f}]\!]_{k,wb} &= \texttt{Await}\ f\ k \\
{}^{s}[\![\texttt{return x}]\!]_{k,wb} &= \texttt{Return}\ x\ wb\ k \\
{}^{s}[\![\texttt{return}^{*}\ \texttt{x}]\!]_{k,wb} &= \texttt{Return}\ x\ \texttt{Nothing}\ k \\
{}^{s}[\![\texttt{return}^{z}\ \texttt{x}]\!]_{k,wb} &= \texttt{Return}\ x\ (\texttt{Just}\ z)\ k \\
{}^{s}[\![\texttt{x:=}V]\!]_{k,wb} &= \texttt{Assign}\ x\ {}^{V}[\![V]\!]\ k \\
{}^{s}[\![\texttt{x:=new}]\!]_{k,wb} &= \texttt{Assign}\ x\ \texttt{New}\ k \\
{}^{s}[\![\texttt{x:=f.get}]\!]_{k,wb} &= \texttt{Assign}\ x\ (\texttt{Get}\ f)\ k \\
{}^{s}[\![\texttt{x:=y!m}(\bar{z})]\!]_{k,wb} &= \texttt{Assign}\ x\ (\texttt{Async}\ y\ m\ \bar{z})\ k \\
{}^{s}[\![\texttt{x:=m}(\bar{z})]\!]_{k,wb} &= \texttt{Assign}\ x\ (\texttt{Sync}\ m\ \bar{z})\ k \\
{}^{s}[\![S_1; S_2]\!]_{k,wb} &= {}^{s}[\![S_1]\!]_{k',wb} \text{ with } k' = {}^{s}[\![S_2]\!]_{k,wb} \\
{}^{s}[\![\texttt{if}\ B\ \{S_1\}\ \texttt{else}\ \{S_2\}]\!]_{k,wb} &= \texttt{If}\ {}^{B}[\![B]\!]\ (\backslash k' \to {}^{s}[\![S_1]\!]_{k',wb})\ (\backslash k' \to {}^{s}[\![S_2]\!]_{k',wb})\ k \\
{}^{s}[\![\texttt{while}\ B\ \{S\}]\!]_{k,wb} &= \texttt{While}\ {}^{B}[\![B]\!]\ (\backslash k' \to {}^{s}[\![S]\!]_{k',wb})\ k
\end{aligned}
$$

$$
\begin{aligned}
{}^{m}[\![m]\!] = (\texttt{m l this wb k} = {}^{s}[\![S_m]\!]_{\texttt{k,wb}}) \\
\text{where } m(\bar{w}) \mapsto S_m \in D \text{ and } \texttt{l} \text{ is the Haskell list that contains} \\
\text{the same elements as the sequence } \bar{w}
\end{aligned}
$$

Figure 3.6: Translation of ABS-subset programs to Haskell AST

```
data B = B :∧ B | B :∨ B | :¬ B | V :≡ V
data V = A Ref | P Ref | I Int
       | Add V V | Sub V V ...
```

Listing 3.4: The syntax and types of the target language. Continuations are wave-underlined. The program/process final result type is double-underlined

The compilation of statements is shown in Fig. 3.6. 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 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, and ${}^{V}[\![V]\!]$ the translation of integer expressions, references or variables. A method definition translates to a Haskell function that includes the compiled body.

```
main, map, reduce :: Method
main [] this wb k =
  Assign node1 New $
  Assign node2 New $
  Assign f1 (Async node1 map [v1])$
  Assign f2 (Async node2 map [v2])$
  Await f1 $
  Await f2 $
  Assign r1 (Get f1) $
  Assign r2 (Get f2) $
  Assign r (Sync reduce [r1,r2]) $
  Return r wb k

map [v] this wb k = ...
reduce [a,b] this wb k = ...

-- Position in the attribute array
[node1,node2,f1,f2,r1,r2,r] = [0..]
```

Listing 3.5: The Haskell-translated running example of MapReduce

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 one object holding a pair of its attribute array and process queue (double-ended) in Haskell `IOVector (IOVector Ref, Seq Proc)`. A cell in 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, i.e. `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, by inspecting the source code (as shown in last line of Listing 3.5).

An `eval` function 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, effectively 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 "trampolines" over 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 queue—meaning the ABS program is either finished or deadlocked. At any point in time, the pair of the scheduler's object queue with the heap comprise the program's state.

Comparison. The described target language is an untyped extract of the canonical HABS backend [Bezirgiannis and Boer, 2016], also described in Section 3.3, with the main difference being that ABS statements are translated to an AST interpreted by `eval` function, while the canonical version compiles statements down to native code, which naturally yields faster execution. However, this deep embedding of an AST allows multiple interpretations of the syntax: debug the syntax tree and have an equivalence result. At runtime, the `eval` function operates in "lockstep" (i.e. executing one CPS statement at a time) whereas the canonical backend applies CPS between release points (`await`, `get` and `return` from asynchronous calls) which benefits in performance but would otherwise make reasoning about correctness and resource preservation for this setup more involved. Another argument for lockstep execution is that we can "simulate" a global Haskell-runtime scheduler (with a N:1 threading model) and include it in our proofs, instead of reasoning for the lower-level C internals of the GHC runtime thread scheduler (with M:N parallelism).

Our target language is also related to *Coroutining Logic Engines* presented in [Tarau, 2011] for concurrent Prolog. These engines encapsulate multi-threading by providing entities that evaluate goals and yield answers when requested. They follow a similar coroutine approach, however, logic engines can produce several results, whereas asynchronous methods can be suspended by the scheduler many times but they only generate one result when they finish.

3.7.4 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. 3.7, 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. 3.7 presents two main differences w.r.t. that in Fig. 3.4 and 3.5. 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 meth-

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

$$(\textsc{Assign})\ \frac{nextObject(h,[\overline{o_m}]) = o_n \quad h(o_n)(\mathcal{Q}) = (\texttt{Assign}\ x\ V\ k', l)\cdot q \quad \texttt{getVal}(h(o_n), V) = v \quad h' = h[(o_n)(x) \mapsto v, (o_n)(\mathcal{Q}) \mapsto (k', l)\cdot q]}{(h,[\overline{o_m}]) \rightarrowtail (h', [\overline{o_{n+1\to m}}] : [\overline{o_{1\to n}}])}$$

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

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

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

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

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

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

$$(\textsc{Return}_A)\ \frac{nextObject(h,[\overline{o_m}]) = o_n \quad h(o_n)(\mathcal{Q}) = (\texttt{Return}\ x\ \texttt{Nothing}\ _, l)\cdot q \quad newQ_{del}([\overline{o_m}], o_n, q) = s \quad h' = h[l \mapsto \texttt{Right}\ h(o_n)(x), (o_n)(\mathcal{Q}) \mapsto q]}{(h,[\overline{o_m}]) \rightarrowtail (h', s)}$$

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

Figure 3.7: Intermediate semantics.

ods m with the adequate parameters. Nevertheless, the rules of the $\rightarrowtail$ semantics correspond with the semantic rules in Section 3.7.1.

Given a list $[\overline{o_m}]$ we use the notation $[\overline{o_{i\to k}}]$ for the sublist $[o_i, o_{i+i}, \ldots, o_k]$, and the operator (:) for list concatenation. In the rules (Async) and ($\textsc{Return}_A$), where the object list can increase or decrease one object, we use the following auxiliary functions. $newQ_{add}([\overline{o_m}], o_n, o_y)$ inserts the object o_y into $[\overline{o_m}]$ if it is new (i.e., it does not appear in $[\overline{o_m}]$), and $newQ_{del}([\overline{o_m}], o_n, q_n)$ removes the object o_n from $[\overline{o_m}]$

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

Figure 3.8: Translation from source to target configurations.

if its process queue q_n is empty. In both cases they advance the list of objects to o_{n+1}.

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

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

In order to reason about the different semantics, we define the translation from runtime configurations $\langle C, h\rangle$ of Section 3.7.1 to concrete Haskell data structures used in the intermediate $\rightarrowtail$ semantics and in the compiled Haskell programs (see Fig. 3.8). 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 Section 3.7.3. 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. The proof can be found in Section 3.7.7.

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)* eval o$_i$ h$_i$ = (res$_i$, l$_i$, h$_{i+1}$), *c) $s_{i+1} = updL(s_i, o_i, l_i)$. Then ${}^{c}[\![(h_1, s_1)]\!]^{-1} \rightarrow^{o_1}_{res_1} {}^{c}[\![(h_2, s_2)]\!]^{-1}_{c} \rightarrow^{o_2}_{res_2} \ldots \rightarrow^{o_{n-1}}_{res_{n-1}} {}^{c}[\![(h_n, s_n)]\!]^{-1}$.*

Note that it is not possible to obtain a similar result about trace completeness since the $\rightarrow$-semantics in Fig. 3.5 selects the next object to execute nondeterministic (random scheduler), whereas the intermediate $\rightarrowtail$-semantics in Fig. 3.7 follows a concrete *round-robin* scheduling policy. The proofs of the theorems is included in Section 3.7.7. 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* [Nipkow et al., 2002] or to generate tests automatically using *QuickCheck* [Claessen and Hughes, 2011].

3.7.5 Resource Preservation

A strong feature of our translation is that the Haskell-translated program preserves the *resource consumption* of the original ABS program. As in [Albert et al., 2015b] we use the notion of *cost model* to parameterize the type of resource we want to bound. Cost models are functions from ABS statements to real numbers, i.e., $\mathcal{M} : S \rightarrow \mathbb{R}$ that define different resource consumption measures. For instance, if the resource to measure is the number of executed steps, $\mathcal{M} : S \rightarrow 1$ such that each instruction has cost one. However, if one wants to measure memory consumption, we have that $\mathcal{M}(new) = \texttt{c}$, where c 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 $\mathcal{T} \equiv A_1 \rightarrow^{o_1}_{S_1} \ldots \rightarrow^{o_{n-1}}_{S_{n-1}} A_n$, 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 $\mathcal{T} = {}^{c}[\![(h_1, s_1)]\!]^{-1} \rightarrow^{o_1}_{res_1} {}^{c}[\![(h_2, s_2)]\!]^{-1}_{c} \rightarrow^{o_2}_{res_2} \ldots \rightarrow^{o_{n-1}}_{res_{n-1}} {}^{c}[\![(h_n, s_n)]\!]^{-1}$ 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 [Albert et al., 2015b]) 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*

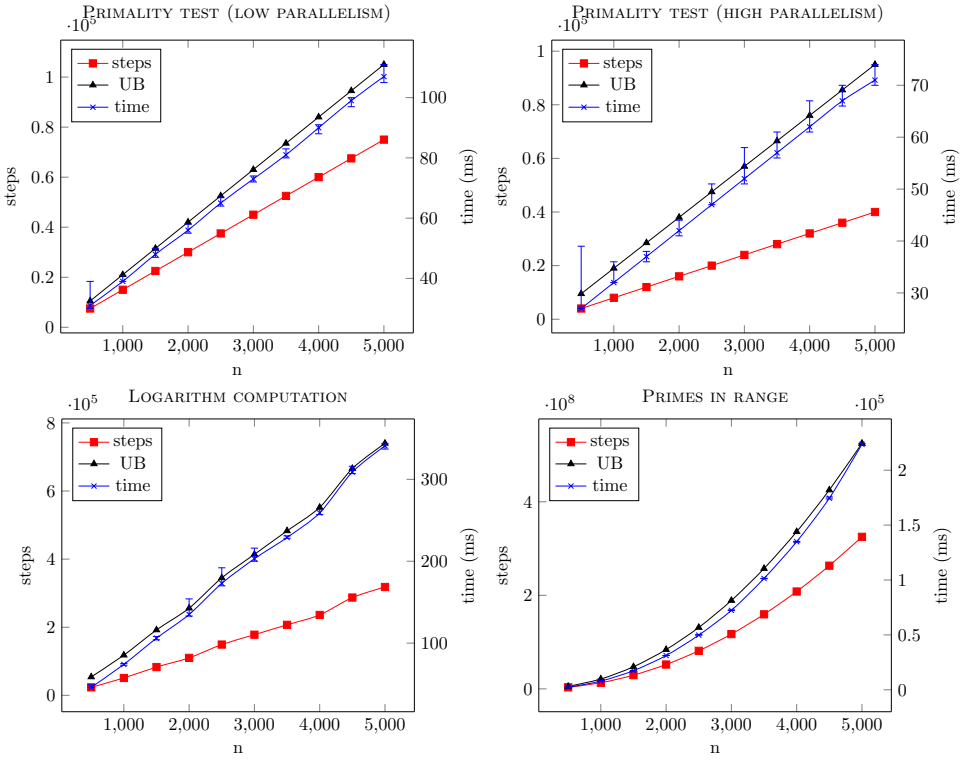


Figure 3.9: Execution steps vs. time (Intel® Core™ i7-4790 at 3.60GHz, 16 GB).

P starting from the main block, restricted to the object o. Then $\mathcal{C}(\mathcal{T}_E, o, \mathcal{M}) \leq UB_{main}()|_o$

3.7.6 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 w.r.t. any concrete cost model $\mathcal{M}$, i.e., both programs execute the same ABS statements in the same order and in the same objects. However, cost models are defined in terms of ABS statements so they are unaware of low-level details of the Haskell runtime environment as β-reductions or garbage collection. Studying the relation between cost models and some significant low-level details of the Haskell runtime in a formal way is an interesting line of future work. In this section we address empirically one par-

ticular topic: the Haskell runtime does not introduce additional overhead, i.e., the execution of one ABS statement requires only a constant amount of work. In order to evaluate this hypothesis, we have elaborated programs[4] with different asymptotic costs and measured the number of statements executed (steps) and their run-time. The *Primality test* computes the primality of a number n: the program creates n objects and checks every possible divisor of n on each object. The difference is that the *low paralellism* version awaits for the result of one divisor before invoking the next check and the *high parallelism* version does not. Both programs have a $O(n)$ cost. The *Logarithm computation* program computes the integer part n logarithms. It has cost $O(n.log\ n)$. Finally *Primes in a range* computes the prime numbers in the interval $[1..n]$, thus having a $O(n^2)$ cost.

We have tested the programs with n ranging from 500 to 5000, running 20 experiments for every value of n, and measured the time. This is plotted in the cross line (right margin) in Fig. 3.9. The plot represents the mode times and the minimum and maximum times as *whiskers*. We have also measured the actual number of steps, represented in the square line (left margin) in Fig. 3.9. These two plots show that the execution time and the number of executed steps grows with a similar rate in all the programs, independently of their asymptotic cost, thus confirming that the compilation does not incur any overhead.

We have also plotted the resource bounds obtained by the SACO tool [Albert et al., 2014a] for the different values of n (triangle line, left margin in Fig. 3.9). SACO can analyze full ABS programs and thus also the subset considered in this section, and allows the selection of the cost model of interest. In this case we have analyzed the original ABS programs using the cost model that obtains the number of ABS statements executed. As can be appreciated, the upper bounds are sound and overapproximate the actual number of executed statements. The difference between the upper bounds and the actual number of statements executed is explained for two reasons. First, the SACO tool considers constructor methods, i.e., methods that are invoked on every new object, so the SACO tool will count a constant number of extra statements whenever a new object is created. However, the main source of imprecision are branching points where SACO combines different fragments of information. A clear example are loops like the one in the *Primes in a range* program. The main loop checks if a number $i \in [1..n]$ is a prime number on each iteration, and this check needs the execution of i statements. In this situation SACO considers that every iteration has the maximum cost (n statements) and generate an upper bound of n^2 instead of the more precise (but asymptotically equivalent) expression $1 + 2 + \ldots + n$.

In the future we plan to extend our formalisations to accommodate full ABS, both in terms of the omitted parts of the language as well as the non-deterministic behaviour of a multi-threaded scheduler, e.g. by broadening our simulated scheduler to non-determinism, and perhaps (M:N) thread parallelism. Another consider-

[4]The ABS-subset experimental programs and measurements together with the target language & runtime reside at `http://github.com/abstools/abs-haskell-formal`.

ation is to relate our resource-preservation result to a distributed-object extension of HABS [Bezirgiannis and Boer, 2016], detailed in Chapter 5; specifically, how the resource analysis translates to network transport costs after any network optimizations or protocol limitations. Finally, we plan to formally relate the ABS cost models used to define the cost of a trace and some of the low-level runtime details of the Haskell runtime like β-reductions, garbage collections or main memory usage. Thus, we could express trace costs and upper bounds in terms closer to the actual running environment.

3.7.7 Proofs and 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 3 (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)**.

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

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

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

  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 $[\overline{o_{n+1 \rightarrow m}}] : [\overline{o_{1 \rightarrow n}}]$, and $\texttt{getVal}(h(o_n), V) = \texttt{getVal}(h_c(o_n), {}^V[\![V]\!])$ because both functions are the same but except from the difference in the languages: syntactic elements versus Haskell terms.

- **(Internal)+(Get)**.

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

  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:

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

  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)**.

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

  where

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

  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)**.

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

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 $h_c = h[(o_m)(\mathcal{Q}) \mapsto {}^{q}[\![Q_m]\!]]$. 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:

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

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 $h_c = h[(o_m)(\mathcal{Q}) \mapsto {}^q[\![Q_m]\!]]$. 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:

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

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 4 (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 3 so we only include the case of (Assign); the other rules follow the same ideas.

- **(Assign)**.

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

The inverse translation of t_A is defined as

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

where ${}^V[\![V]\!]^{-1} = V'$, ${}^s[\![k']\!]^{-1} = S$, ${}^q[\![q]\!]^{-1} = Q$, 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:

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

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 5 (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)**.

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

If $k = \texttt{Assign}\ x\ V\ k'$ then `res` will be `Assign lhs` ${}^{V}[\![V]\!]$ `k'` where `lhs` is the position in the vector `attrs` of the variables `x`. Therefore the `case res of` expression will execute the following branch:

```
7 Assign lhs (Val x) k' -> do
8   (attrs `V.write` lhs) =<< (getVal x)
9   updateObj $ Left k'
10  return (res,
11          [this],
12          h)
```

The heap is updated to store the value of the expression `x` using the vector operators `V.write`. The concrete value of the expression `x` is obtained using the inner function `getVal :: V -> IO Int`. Then 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 `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 $l \equiv [o_n]$.

- **(New)**.

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

If $k =$ `Assign` x `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:

```
13 Assign lhs New k' -> do
14   (attrs `V.write` lhs) $ newRef h
15   updateObj $ Left k'
16   initAttrVec <- V.replicate 10 (-1)
17   (objects h `V.write` newRef h) (initAttrVec, S.empty)
18   h' <- incCounterMaybeGrow
19   return (res,
20           [this],
21           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 $h' =$ `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 $l \equiv [o_n]$.

[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.

- **(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.

- **(Await I)**.

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

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)**.

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

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)**.

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

  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$_A$)**.

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

  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$_S$)**.

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

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

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 5. □

Proof of Theorem 1 (Trace soundness)

Proof. By induction on the number of `eval` steps using Lemmas 6 and 4. □ □

Auxiliary definitions and results for bound preservation

In order to prove the preservation of the bounds obtained in [Albert et al., 2015b] we need to prove that for any trace $\rightarrow$ there is and equivalent trace using the semantics $\rightsquigarrow$ considered in [Albert et al., 2015b]. 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 3.7.1 and its semantics in Fig. 3.4 and 3.5 are a simplified version of those in [Albert et al., 2015b]. 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 [Albert et al., 2015b] 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 [Albert et al., 2015b] rewrites states $St \rightsquigarrow St'$.

We will consider a slight variation of the operational semantics in [Albert et al., 2015b] 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 [Albert et al., 2015b]. To simplify the results, we will assume that the decorations of the $\rightsquigarrow$-steps use the syntax presented in Section 3.7.1.

In order to prove Theorem 2 we will define a translation from configurations as defined in Section 3.7.2 to states in the semantics in [Albert et al., 2015b]. 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)**.**

$$\begin{array}{rcl}
\|\langle C, h\rangle\| & = & \{ob(n, _, h(n), a, t) | (n : Q) \in C, (a, t) = \|Q\|_q\} \cup \\
& & \{ob(o, _, \epsilon, \epsilon, \emptyset) | o \in Dom(h) \setminus objs(C)\} \cup \\
& & \{fut(fn, v) | fn \in futs(h), h(fn) = v\} \\
\\
\|\epsilon\|_q & = & (\epsilon, \emptyset) \\
\|(S; l) \cdot (S_1; l_1) \cdot \ldots \cdot (S_n; l_n)\|_q & = & (\langle [\texttt{ret} \mapsto l], \|S\|_s\rangle, \\
& & \{\langle [\texttt{ret} \mapsto l_1], \|S_1\|_s\rangle, \ldots, \langle [\texttt{ret} \mapsto l_n], \|S_n\|_s\rangle\} \\
\\
\|\epsilon\|_s & = & \epsilon \\
\|\texttt{x:=}V; S\|_s & = & \texttt{x:=}\|V\|_v; \|S\|_s \\
\|\texttt{x:=new}; S\|_s & = & \texttt{x:=new}; \|S\|_s \\
\|\texttt{x:=f.get}; S\|_s & = & \texttt{x:=f.get}; \|S\|_s \\
\|\texttt{f:=x!p}(\bar{z}); S\|_s & = & \texttt{call(m,p(x,}\bar{z}\texttt{,f))}; \|S\|_s \\
\|\texttt{f:=p}(\bar{z}); S\|_s & = & \texttt{call(b,p(this,}\bar{z}\texttt{,_))}; \|S\|_s \\
\|\texttt{await f}; S\|_s & = & \texttt{await f}; \|S\|_s \\
\|\texttt{return x}; S\|_s & = & \texttt{return x}; \|S\|_s
\end{array}$$

where $\|V\|_v$ is the straighforward translation of variables, references and integer expressions.

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 [Albert et al., 2015b] *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 7 (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)* `(res_i, l_i, h_{i+1}) = eval o_i 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 $S_i \equiv res_i$)

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

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 9 (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 10 (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 8. *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)**.

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

The translation of S_1 is

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

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 (1) in [Albert et al., 2015b], reaching $\|B\|$:

$$(1)\frac{v = eval(\|V\|_V, h(n), [\texttt{ret} \mapsto l])}{\begin{array}{l}\{ob(n, _, h(n), \langle[\texttt{ret} \mapsto l], \texttt{x:=}\|V\|_V; \|S\|_s\rangle, Q_{tr})|R\} \rightsquigarrow^n_{\texttt{x:=}\|V\|_V} \\ \{ob(n, _, h(n)[x \mapsto v], \langle[\texttt{ret} \mapsto l], \|S\|_s\rangle, Q_{tr})|R\} \equiv \|B\|\end{array}}$$

Note that *eval* is the function in [Albert et al., 2015b] that computes the value of simple right-hand sides of assignments, so it behaves exactly like $\texttt{getVal}(h, V)$.

- **(Internal)+(New)**.

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

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 [Albert et al., 2015b], 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 [Albert et al., 2015b]:

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

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 [Albert et al., 2015b]:

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

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.

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

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 [Albert et al., 2015b]:

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

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)**.

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

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 [Albert et al., 2015b], reaching $\|S_2\|$:

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

$\|D\|^x_{sync}$ 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 [Albert et al., 2015b] 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$_A$)**.

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

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 [Albert et al., 2015b]:

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

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 9. *If $\mathcal{T} = A_1 \rightarrow^{o_1}_{S_1} A_2 \rightarrow^{o_2}_{S_2} \ldots \rightarrow^{o_{n-1}}_{S_{n-1}} A_n$ 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 8. □ □

Lemma 10. *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 [Albert et al., 2015b]), since only the steps decorated with a statement (i.e., different from ϵ) contribute to the cost. □ □

Proof of Theorem 2 (Bound Preservation)

Proof. Straighforward by Lemma 7 and Theorem 3 from [Albert et al., 2015b]. □

3.8 Case Study on Preferential Attachment

We decided to use HABS in a real-world case study of generating network graphs in parallel. The preferential attachment is a special class of network generation where new nodes are sequentially introduced to the network and they attach preferentially to existing nodes. Such generation process is commonly found in social networks.

The Barabasi-Albert model [Barabási and Albert, 1999] is written to generate scale-free networks using the preferential attachment mechanism. However the sequential mechanism used in this PA model makes it an inefficient algorithm. Other existing parallel approaches, on the other hand, suffer from either changing the original model or explicit complex low-level synchronization mechanisms. We develop a parallel version of the PA model in ABS that stays at a high-level, thanks to the actor model abstraction.

To implement the PA model in ABS, we first extend the ABS language with support for promises. In general, this feature can cause complicated and hard-to-verify programs and thus the programmer should use the feature in a disciplined manner, such as provided by our model (which restricts the model to single write access), to avoid race conditions.

Apart from a functional layer which includes algebraic data types and pattern matching, the implementation additionally features global arrays as a mutable data structure shared among objects which fits well in the multicore setting to decrease the amount of costly message passing, and also to simplify the model. To achieve this we utilized the Foreign Language Interface extension to ABS (shown in section 3.2.4). The ABS code for the algorithm maintains a global, mutable, $O(1)$, boxed array: each array-cell is an ABS promise coupled with a set of active objects (their thread references) as "listeners". An ABS process will suspend its execution until the future of the array cell is resolved; the active object that resolves the future will inform the set of listeners to wake up the corresponding suspended processes. This extension of promise-arrays is integrated naturally in the ABS ecosystem through the `await` on-boolean-condition. Finally, we use a foreign-imported random-number library with each active object having each own, separate random-number generator for performance reasons.

3.8.1 Results

We ran the program of the PA-based generation of networks in ABS2Haskell based on the proposed approach on SURFsara cluster on a 16 core processor 2.30 GHz (Intel Xeon CPU E5-2698 0) with 128GB of memory [6].

The program is verified using a set of test cases (e.g. checking for the resolution of *all* edges of the graph and checking duplicates for the final graph). According to this experiment, the degree distribution of the graphs generated by our proposed method follows a power-law degree distribution. In Figure 5.2, the performance and the scalability of the program is depicted for different input parameters. The performance of the program is good in comparison with the performance of the efficient sequential implementation of the PA in HABS.

Looking at the performance results, one point worth mentioning is the super-linear speedup observed when going from 1-core to a 2-core execution for any of the 4 distinct runs. We speculate that this can most likely be attributed to the great

[6] SurfSARA `http://surf.nl`

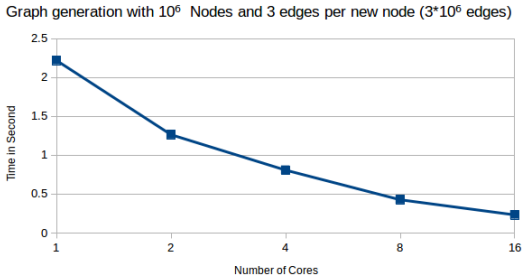


Figure 3.10

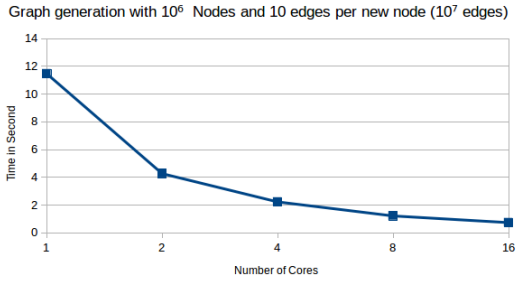


Figure 3.11

effect a multi-level CPU cache can have on a multicore setup. Specifically for our case and granted our SURFSara experimentation system, a doubling in number of cores leads to the doubling of the size of L1 and L2 cache (the shared L3 cache stays the same). This results to less overall cache misses on a 2-core setup, which greatly adds to the performance, hence the super-linear speedup. However, this effect is only clearly observable when transitioning from 1-core to 2-core; after 2 cores, the parallel threading overhead overshadows any larger-cache benefit. Still, this remains just a speculation; we are planning to investigate more on the reason and the impact a cache behaviour can have over the PA graph generation.

3.9 Related Work

Over the years after the appearance of the actor model there have been numerous programming languages and special libraries that (try to) implement it. Note that some other concurrency-based languages that relate more to theory and modeling

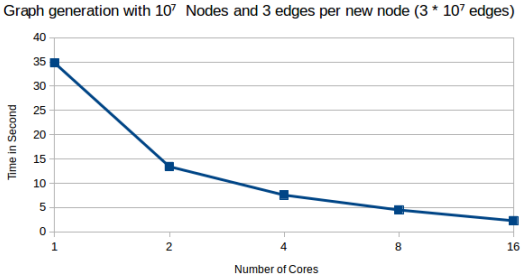


Figure 3.12

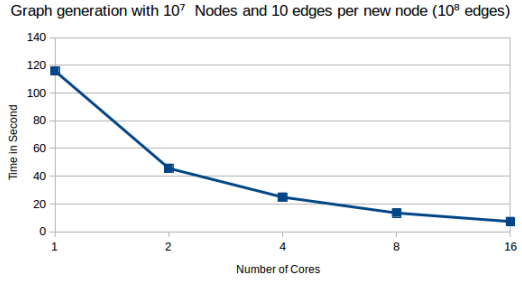


Figure 3.13

are discussed in section 2.9. Here we focus on actor-based languages for practical purposes. Arguably, the most well-known actor programming language is Erlang, a dynamically-typed, (non-purely) functional programming language which can be bytecode-interpreted by its VM (BEAM); thus Erlang code is very portable, since the same bytecode can be used by any computer system (operating system and cpu architecture) where the Erlang VM runs on. Erlang, like ABS, *disallows* any shared-memory access for concurrent programs: any data exchange has to strictly go through message passing. Erlang can also support a kind of "object-oriented" storage inside the actor (active object) by means of the so-called "process dictionary", a private storage for each process. These processes are Erlang's actors, built into the language runtimes as lightweight (also called green) threads; HABS active-objects are instead coroutines (even more lightweight). Erlang supports Simultaneous Multi-Processing (SMP) with preemptive scheduling which automatically load-balances its processes (actors) over the system's CPU cores, as does HABS (for its COGs) with its GHC Haskell runtime. Erlang does not have any notion of a COG or cooperative scheduling. We defer the discussion of the distributed-computing part of Erlang on the more related section 5.5.1 of the distributed HABS implementation.

Although strictly not a language but a library, Akka (`http://akka.io`) has become relatively famous to the Scala and Java communities for introducing the actor-model type of concurrent and distibuted computing to the JVM ecosystem. Unlike ABS, the actors are *not* protected from race conditions, because there exist still the possibility of shared-memory, "leaked" access between actors via the underlying heap, although such thing is discouraged in favour of the "safer" message passing. Still though, there exist a source of "unsafety" since messages cannot be guaranteed to be immutable. By default Akka's actors are untyped like Erlang's messages, however unlike Erlang, changing the behaviour of the actors (i.e. an actor to decide dynamically at runtime to receive a different message) has to be explicit and arguably more complicated through the use of `become()`/`unbecome()` statements that perform hot-code swapping of the actor's implementation. For our case, ABS lacks builtin support of the Actor model for determining how to receive the next message (e.g. become in Akka's Untyped Actors) but such behaviour can be emulated programmatically for ABS and for Akka's *typed actors*. Typed actors is an experimental addition to the Akka library and looks much closer to the ABS's active objects where asynchronous communication is encapsulated behind method calls. Akka does not support awaiting on booleans, but offers many practical features borrowed from Erlang and other languages, e.g. supervisors, streams, routers. Finally, Akka is constrained by a threadpool (since JVM threads are expensive) for supporting Simultaneous Multi-Processing with preemptive scheduling for its active objects (actors); as such, an Akka actor system is prone to process starvation or even deadlock, by not correctly utilizing the event-based mechanism of the library.

Pony [Clebsch et al., 2015] `http://ponylang.org` is a relatively recent concurrent programming language which with a C-written library and runtime for support of the actor model. As such, it offers a strong connection to C with an FLI. Pony

adds an elaborate type-system based on reference-capability security: a capability is roughly an object reference together with attached access rights for the caller of the object. Pony offers both nominal and structural subtyping for its objects. Unlike ABS, methods cannot be called both synchronously and asynchronously: their (a)synchronicity is declared at their method-definition. Asynchronous methods in Pony do not return a result, so there is no implicit bi-directional communication encapsulated behind the method call — a specific trait of the so-called active object pattern. As such, there is no built-in await mechanism (neither or futures nor booleans): the caller can only pass a function callback (closure) to be executed by the callee when the method is completed. However, there is limited support for cooperative multitasking in Pony through promises (read-write futures) and streaming. Similar to Akka, the Pony runtime employs a thread-pool – the size defaults to the number of CPU cores — for SMP preemptive multitasking which means the problem of process starvation still remains. HABS, however, solves this by utilizing Haskell's lightweight threads (with an M:N threading model of GHC's runtime). Finally, Pony does not have algebraic datatypes.

Encore [Brandauer et al., 2015] is a higher-level actor-based programming language which builts on top of the Pony runtime system, thus its runtime characteristics match those of the Pony language. Encore offers almost all language features of Pony, and adds support of bi-directional communication, i.e. asynchronous method calls return a Future. Furthermore, Encore adds an await mechanism for a non-blocking read of the Future value, i.e. the `await`-caller can be activated on other methods (processes in ABS). Encore supports the so-called "future-chaining" which allows to non-blocking map a function to a future container (like Haskell's Functor). Encore goes a step further than Pony by allowing the inline of C code inside Encore program code. The language, as of currently, lacks awaiting on booleans and algebraic datatypes.

Not relating to the actor model but to our target language, O'Haskell [Nordlander, 2002] is an attempt to bring the object-oriented paradigm to Haskell. Unfortunately O'Haskell is a separate language inspired by Haskell and cannot utilize already-existing Haskell code. The offered subtyping is structural (compared to ABS and HABS nominal), which makes it easier to augment the type inference of Haskell's type system. Similar to Pony, O'Haskell has support for "reactive" agents, which are event-driven objects that do not return a result (future). On a different direction, [Kiselyov and Laemmel, 2005] implement the object-oriented paradigm in a library, using purely Haskell constructs. Although the authors detail certain performance penalties for doing this "shallow-embedding" of OO in Haskell, the end-result has a very flexible and powerful type system, offering both structural subtyping and parametric polymorphism.