Abstract Behavioral Specification: unifying modeling and programming

Proefschrift

ter verkrijging van

de graad van doctor aan de Universiteit Leiden

op gezag van de Rector Magnificus prof. mr. C. J. J. M. Stolker,

volgens besluit van het College voor Promoties

te verdedigen op dinsdag 17 april 2018

klokke 15.00 uur

door

Nikolaos Bezirgiannis

geboren te Thessaloniki, Griekenland,

in 1987

PhD committee

Promotor:	Prof. dr. F.S. de Boer	
Co-promotor:	Dr. C. P. T. de Gouw	Open Universiteit
Other members:		
	Prof. dr. A. Plaat	
	Prof. dr. F. Arbab	
	Prof. dr. E. B. Johnsen	University of Oslo
	Prof. dr. T. van der Storm	Rijksuniversiteit Groningen

The work reported in this thesis has been carried out at the Center for Mathematics and Computer Science (CWI) in Amsterdam and Leiden Institute of Advanced Computer Science at Leiden University, under the auspices of the research school IPA (Institute for Programming research and Algorithmics). This research was supported by the European FP7-610582 project ENVISAGE on Engineering Virtualized Resources.

Contents

Chapter 1

Introduction

The latest advancements in technology and economic progress led to the ubiquity of computers (hardware and software) in our daily lives. Currently, computer systems are present (embedded) in our phones, watches, automobiles, and even coffee machines and lamps; the future seems to be even more intrusive with computers appearing inside our clothes and under our skin. This enormous information-gathering from all these computers (sensors) demands an analogously-large computing power to process this information in a fast or timely manner.

The future does not look to be so bright, however, when it comes to hardware's raw processing power. For long, it has been established that Moore's law is constrained by the speed of light, that is, there is a limit on how fast the information can flow (and thus be processed) inside a computer system. For mainly this reason, hardware manufacturers have been trying to make the sizes of transistors (i.e. raw processing power) and the distances between them smaller and smaller through these years of development. Yet, there are indications that we have reached this time another limit, where manufacturing at atomic (or even subatomic) levels of transistor size is unstable to produce circuits and thus not economically-viable at large scales (production yield).

For the last decade, this sole reason has driven manufacturers to turn this time to parallel computing for keeping up with the Moore's law, by packing more and more multiple (and otherwise independent) computing resources (CPU cores) to form a (single) larger computing system (i.e. the multicore CPU). This revolution has already reached the mainstream consumer hardware where, as of 2017, a common smartphone can contain up to 10 cores and an affordable desktop machine up to 32 cores. The underlying idea behind multicore computing is that the performance improves when we split up the workload into multiple (equal) parts and process these *in parallel* to produce back the same result. For many common workloads (beside graphics computing like GPUs), the cores have to *communicate* sometimes with each other to perform a single task. This communication (information flow) between the cores is, once again, constrained by the speed of light. There are even certain workloads where the cores' communication is such an overhead that it is faster to run the computation sequentially (i.e. with a single core). Although the industry yet seems optimistic in coming up with increasingly larger counts of CPU cores, there exists the eventual cut-off point, where the limit of light speed will deem CPUs with million or billion cores unsuitable. It has become a recent topic of wide discussion whether the Moore's law will still hold for the latest hardware developments[1].

Another recent advancement that has contributed to the performance of computer processing is the creation of the "Cloud" infrastructure. Although the similar distributed computing paradigm has been investigated long before the Cloud, it has only been the past decade where such online offerings of hardware infrastructure have become economically viable. Furthermore, the nowadays dominance of "Anything"-as-a-service has contributed to the recent popularization of cloud systems, because of the easy scaling (vertical or horizontal) that the Cloud offers. However, distributed (and cloud) computing hardware is restricted from the same constraint of the speed of light. In fact, the importance of this constraint is many-fold magnified since the individual computing processors of the Cloud are often geographically sparsely located: not interconnected through silicon (as in multicores) but through longer networks (e.g. ethernet cables). In many cases the communication overhead in the Cloud is so profound that poses beside the physical limitation (lightspeed), an algorithmic problem: how the computation can be distributed (split-up) without being slowed down by the communication overhead. The challenge arises: how can we utilize these cloud resources optimally?

These limitations in hardware's raw computing power have moved the attention instead to software, so as to "squeeze" (optimize) the last amount of performance gain possible. Coupled with the programming paradigm shift that the multicore revolution brought — where coming up with parallel algorithms and coding them can often be hard and erroneous — led to a huge burden put to the software development for being fast while all the while being increas-

[1]The Economist - The End of Moore's law `http://www.economist.com/blogs/economist-explains/2015/04/economist-explains-17`
ElectronicsWeekly - Is Moore's law still the law? `https://www.electronicsweekly.com/news/moores-law-still-law-2017-09/`

ingly complex. To reach optimal performance on the computing infrastructure where the software is deployed, the software needs to be aware and be able to control (to some extent) the utilizations of the underlying resources.

Software modeling is a relatively-recently introduced concept to tackle mainly the “pillar” of complexity. It achieves this, by allowing the user to abstract from implementation details and instead focus on the functional correctness of the software. Modeling deals with constructing a higher-level abstraction (model) of the software even before it is actually constructed. A model is governed by a set of formal (concrete) rules which makes it difficult by-definition to introduce errors into the model. Furthermore, this rigor can help reason in a high mathematical level about the internals of the software, and as a result, faults in the design and infrastructure of the software can be detected early on. Often, these formal rules are written as computer programs themselves (proof assistants or theorem provers) which allows *automatic* checking of a model against a set of rules, instead of manually proving its correctness. Little has been done, however, to achieve performance in software modeling comparable to a (lower-level) optimized executable program. There have been some previous efforts [Long et al., 2005, Moreira et al., 2010] to generate (efficient) code from a model and later include it as part of a program, but these do not take the available computing resources into account (and thus cannot exploit them optimally). Furthermore, the integration of such generated code in production code has often been omitted or under-specified. Hence, the question arises: how can we optimize performance of software taking aspects of the available computing resources into account, while still remaining at the high-level of abstraction that is crucial for model-based approaches?

Summarizing, the general trend in programming languages is to move away from explicit implementation details and instead focus on abstraction and code portability (e.g. Java) through high-level formalisms. The software technology is trying to “catch-up” with the hardware developments, but this requires the explicit control of the hardware resources and its optimal usage. **The main challenge that arises is how we can abstract away from implementation details, but still manage the hardware resources at a sufficient abstraction level, so that we can benefit from the underlying performance. In this thesis, our main contribution is to address this challenge by constructing a language to write software which can take advantage of recent hardware developments (multicore, cloud) without many compromises in the levels of abstraction.**

The language discussed in this thesis is a modeling language that engages both pillars of software-engineering, namely complexity and more-importantly performance of execution. To achieve this, we aim to provide an interface of inter-operation between the model, the production code and the hardware infrastructure where the software runs on. Besides multicore hardware, we also investigate in running the modeling language in the modern distributed (cloud) computing systems.

1.1 Why ABS

We base our modeling language upon the Abstract Behavioral Specification language (ABS), with its development starting in to 2006 [Johnsen et al., 2006]. Even before that, the ABS language is the continuation of the high-level, concurrent Creol modeling language which is in-turn born out of of the well-known first-ever object-oriented programming language SIMULA, that goes back as early as 1965.

ABS is generally regarded as a modeling language. A modeling language differs from a programming language in that its *primary* goal is not to (easily) construct a software product; a modeling language's purpose is merely to help the user lay down information and structure it at one's will. This structured information (model) may or may not later act as a "vehicle" for constructing software. It can still be the case that a model is solely used for the purpose of brainstorming, idea exploration, experimentation, simulation, or even (human) communication. In this respect, models are usually left abstract or even incomplete; this is aided from the fact that a modeling language is usually governed by a small set of well-defined rules to express the information in a high-level as possible. Moving on, ABS is executable — compared for example to the widely-known modeling language UML — since there is a "mechanized" way to interpret its semantics as transition rules (i.e. an operational semantics) and thus attach a "meaning" to every (well-constructed) model. The question arises again as to how then an *executable* modeling language differs from a programming language which also attaches meanings (semantics) to a program (instead of a model). The answer lies in the separation of their purposes: a programming language aims to generate (fast) production code, an executable modeling language only generates code for the purpose of model reduction, visualization and interactive feedback of information. Although performance of execution is not a primary goal, it can become important if the modeler wants to execute larger or more complicated models and interact with them in a timely manner.

Users of a modeling language (modellers) are generally not expert programmers. ABS aims to stay familiar to the average user by supporting a "friendly" object-oriented programming layer which resembles that of Java. ABS offers a functional layer but unlike other, fully-featured functional programming languages, the language has arguably a smaller learning curve; this is because on one hand its functional features are minimal, and on the other hand, the connection with the object-oriented, imperative world is simpler, compared to monads, type & effect systems, or uniqueness types of other languages.

To further accommodate the average modeler, the ABS ecosystem provides a plethora of development tools: an interactive development environment in the Emacs text editor, a developer plugin for eclipse, interactive debugger and method-call visualizer.

The grammar (syntax) and operational semantics (meaning) of the ABS language are well defined using formal method techniques: in this way the documentation of the language becomes more clear and precise, and more, importantly it enables the rigorous analysis of the language. In fact, many analysis and verification tools have been developed over the course of the years for the ABS language, ranging from termination analysis [Albert et al., 2013], resource analysis [Albert et al., 2015a], deadlock analysis [Giachino et al., 2014], to monitoring [Boer et al., 2013, Wong et al., 2015] theorem proving and full-blown verification [Din et al., 2015, Din et al., 2017].

Commonly in software, and in engineering in general, concurrency and parallelism are two concepts which are both difficult to grasp as well as implement. A major challenge in the design of modeling languages is an appropriate development of a concurrency model. ABS adds support for concurrency and inherent parallelism to the object-oriented paradigm. More specifically, the ABS language combines the Actor model formalism with the notion of the *object* to create the *active object*: the communication to an active object can be as well asynchronous and is encapsulated behind the usual method calls. The language's concurrency model goes a step further and introduces its main, and characteristic, feature of *cooperative scheduling*, also known as (semi-)coroutines. In such a setting, active objects form groups (the so-called Concurrent Object Groups); all active objects inside a group share their computing resources (i.e. thread of execution). A running object can programmatically decide to deliberately yield its control so as another object of the same group can execute, i.e. explicit cooperation which is in contrast to the usual preemption of thread mechanisms.

ABS' concurrency model avoids dangerous programming idioms such as threads and lock mechanisms. The immutability of datastructures in the purely-functional layer together with the notion of future values (write-once placeholders which will be computed in the "future") leads to less race conditions. The fields of an object can only be private, which avoids incidents of pointer aliasing. Lastly, the "yielding of control" of cooperative scheduling happens in explicit places in the program, which makes it more clear on which are the concurrent interleavings of that program. You can find out more about the concurrency model offered by ABS at Section 2.7.

The challenges that we faced during the development of our modeling language include finding the right programming constructs to translate the model to, executing the model through a fast runtime, and showing that the resulting executed model conforms still to the set of rules laid out by the modeling language (i.e. proving correctness). Besides having a generally efficient ABS implementation, we were faced with the implementation of the “cooperation” feature of ABS which is arguably difficult to implement. We try to address this difficulty by developing an efficient runtime environment for ABS.

1.2 Targetting Haskell

To execute the proposed ABS modeling language we translate it to lower-level Haskell program code. Haskell ([Peyton Jones, 2003]) is a general-purpose programming language that first appeared in 1987; its name derives from the mathematician Haskell Curry. Unlike most existing programming languages, designed by a single person or company, Haskell was designed by a committee of academicians for the purpose of “agreeing on a common (lazy functional) language” (from the talk of Simon Peyton-Jones: Escape from the ivory tower: the Haskell journey). Haskell differs from other functional languages since it is *purely* functional: functions play a key role, but they cannot contain any side-effects. This permits the user to “make better sense” of the program's code through equational reasoning and referential transparency. Still, programming completely without side-effects can be a burden or in certain cases impossible — e.g. interacting with the real-world has side-effects — and for this reason Haskell introduces the concepts of Monads (borrowed from Category Theory) and monadic programming to allow side-effects in the language but without breaking purity: there is a clear distinction at the type-level between purely functional and monadic (side-effectful) code. For this reason the type-system of Haskell has been regarded as a very strong static type-system, with other reasons being the support for parametric polymorphism, class-constrained (ad-hoc) polymorphism, type-level programming, datatype-generic programming [Gibbons, 2007], and a limited form of dependently-typed programming [McBride, 2000]. The semantics of Haskell is *by default* call-by-need (also known as lazy). Compared to the commonly-found strict semantics (call-by-value and call-by-reference), Haskell expressions and their sub-expressions will only be evaluated at the specific part that is required by the computation.

Furthermore, unlike the similar call-by-name semantics, lazy semantics will avoid re-computing already evaluated (sub)expressions, which leads to better sharing. Last, lazy semantics admits more expressive power for the language (e.g. when dealing with infinite data structures). Still, the language allows for partially (in places) introducing strictness which may improve the program's performance — most functional languages are strict by-default and optionally lazy.

The choice of Haskell was made since it provides language features that closely match those of the functional layer of ABS, and also certain runtime facilities that make the translation of ABS more straightforward. First of all, both languages offer a purely-functional layer: whereas ABS restricts the mixing of pure and impure code at the syntactic level, Haskell achieves this instead on the type-level. Furthermore, their type-systems share certain commonalities, that is algebraic datatypes with support for parametric polymorphism, ad-hoc polymorphism through ABS interfaces - Haskell's typeclasses. Finally, the module system of both language is quite similar; in fact, the ABS module system was inspired from that of Haskell.

The Haskell type system has been formalized in [Sulzmann et al., 2007, Eisenberg, 2015]. However, the operational semantics of Haskell, and specifically that of the GHC Haskell compiler is hypothetical (not been proven correct yet) as the author say: "It is hypothetical [the semantics] because GHC does not strictly implement a concrete operational semantics anywhere in its code. While all the typing rules can be traced back to lines of real code, the operational semantics do not, in general, have as clear a provenance." Still, since both languages are very similar and stay on the same (high) level of abstraction, it enabled us to prove the correctness and resource preservation of the translation of a subset of ABS to a subset of Haskell (with continuations) which is detailed in Section 3.7.

At the runtime side, the canonical Glasgow Haskell Compiler (GHC) provides a fast and well-tested runtime system where we base the concurrency mechanisms of ABS upon. GHC's features support such as first-class continuations, lightweight (green) threads, load-balancing of threads to multicores for automatic parallelism gain (also known as the M:N hybrid thread model), parallel garbage collection, STM-based datastructures (software transactional memory) among others allowed us to straightforwardly express and thus implement the ABS concurrency abstractions, and more importantly the cooperative scheduling of ABS, in terms of Haskell constructs.

Finally, albeit not directly related to Haskell as the target language, Haskell was chosen as the host language to write the ABS-to-Haskell transcompilation phase, since Haskell is arguably regarded as one of the best languages to write compilers, reasons ranging from the support for brevity through algebraic datatypes, pattern matching, recursion to compilation's safety and correctness provided by the language's elaborate & strong type system.

It is worth noting that we opted against using Haskell directly, but only through a translation. Although Haskell can be very expressive and safe, e.g. monads, its user learning curve is steep with many concepts rooted in the category theory of mathematics, e.g. again monads. Furthemore, these concepts are yet to reach a status of mainstream, so the average user that writes software programs is most likely impervious to them.

Through the translation of ABS to Haskell, we manage to contribute also to the ecosystem of Haskell:

- a Haskell runtime library to express cooperative scheduling.
- a methodology of providing the object-oriented paradigm for Haskell, which Haskell normally lacks, as the consequence of implementing it in our ABS translation.

1.3 Validation

This work has been carried out in the context of the Envisage Project. The ENVISAGE project is a EU-funded project for:

> The development of a semantic foundation for virtualization and service-level agreements (SLA) that goes beyond todays cloud technologies. This foundation makes it possible to efficiently develop SLA-aware and scalable services, supported by highly automated analysis tools using formal methods. SLA-aware services are able to control their own resource management and renegotiate SLA across the heterogeneous virtualized computing landscape.

Our work was validated on two case studies: an industrial case study of the cloud services offered by the SDL-Fredhopper company `https://www.fredhopper.com/` and a case study on the Preferential Attachment problem of dynamics, which is concerned with the efficient generation of social-network-like graphs.

1.4 Outline

Chapter 2 *Abstract Behavioral Specification* (ABS) [Johnsen et al., 2010a] is a formally-defined language for modeling actor-based programs. An actor program consists of computing entities called *actors*, each with a private state, and thread of control. Actors can communicate by exchanging messages asynchronously, i.e. without waiting for message delivery/reply. In ABS, the notion of actor corresponds to the *active object*, where objects are the concurrency units, i.e. each object conceptually has a dedicated thread of execution. Communication is based on asynchronous method calls where the caller object does not wait for the callee to reply with the method's return value. Instead, the object can later use a *future* variable [Flanagan and Felleisen, 1995, Boer et al., 2007] to extract the result of the asynchronous method. Each asynchronous method call adds a new *process* to the callee object's process queue. ABS supports *cooperative scheduling*, which means that inside an object, the active process can decide to explicitly suspend its execution so as to allow another process from the queue to execute. This way, the interleaving of processes inside an active object is textually controlled by the programmer, similar to coroutines [Knuth, 1973]. However, flexible and state-dependent interleaving is still supported: in particular, a process may suspend its execution waiting for a reply to a method call.

Chapter 3 Whereas ABS has successfully been used to model [Wong et al., 2012], analyze [Albert et al., 2014a], and verify [Johnsen et al., 2010a] actor programs, the "real" execution of such programs has been a struggle, attributed to the fact that implementing cooperative scheduling efficiently can be hard (common languages as Java and C++ have to resort to instrumentation techniques, e.g. fibers [Srinivasan and Mycroft, 2008]). This led to the creation of numerous ABS backends with different cooperative scheduling implementations:[2] ABS$\rightarrow$Maude using an interpreter and term rewriting, ABS$\rightarrow$Java using heavyweight threads and manual stack management, ABS$\rightarrow$Erlang using lightweight threads and thread parking, ABS$\rightarrow$Haskell using lightweight threads and continuations.

Implementing cooperative scheduling can be non-trivial, even for modern high-level programming languages (e.g. Java, C++) because of their stack-based nature. A recent relevant technology is to use *fibers* [Srinivasan and Mycroft, 2008], which adds support for cooperative threads by instrumenting low-level code (commonly via bytecode manipulation) to save and restore parts of the stack. We instead opted for source-to-source translating ABS programs to Haskell, a functional language with language-level

[2]See `http://abs-models.org/documentation/manual/#-abs-backends` for more information about ABS backends.

support for coroutines, based on the hypothesis that a high-level translation serves as a better middleground between execution performance and most importantly semantic correctness. Our transcompiler translates ABS programs to equivalent Haskell-code, which is then compiled to native code by a Haskell compiler and executed. Prior alternative approaches for executing ABS have been an Erlang translator, that utilizes Erlang's preemptive lightweight processes to simulate cooperative threads, and a Java translator, that manages a global dynamic pool of heavyweight threads.

Furthermore, we present and discuss a formal translation of a actor-based language with *cooperative scheduling* (a subset of ABS) to the functional language Haskell. Here we make use of a different, more high-level translation of ABS to Haskell than the translation implemented in the ABS→Haskell backend. This formal translation is proven correct with respect to a formal semantics of the source language and a high-level operational semantics of the target, i.e. a subset of Haskell. The main correctness theorem is expressed in terms of a simulation relation between the operational semantics of actor programs and their translation. This allows us to then prove that the resource consumption is preserved over this translation, as we establish an equivalence of the cost of the original and Haskell-translated execution traces. Finally, the method that was developed is general but applied only to a subset of ABS; for future work we consider to apply this method for all ABS constructs for formally verifying the complete ABS language.

Chapter 4 In this chapter we discuss an extension of ABS to write software that can programmatically take control of its computing (hardware-virtualized) resources. This type of programming which we name "resource-aware" programming differs from the usual emulation or hardware description languages, such as Verilog, because it does not focus on the design of (new) hardware but on how software can take advantage and be "aware" of the underlying hardware. We construct an integrated tool-suite for the simulation of software services which are offered on the Cloud hardware. The tool -suite uses the Abstract Behavioral Specification (ABS) language for modeling the software services and their Cloud deployment. For the real-time execution of the ABS models we use a Haskell backend which is based on a source-to-source translation of ABS into Haskell. The tool-suite then allows Cloud engineers to interact in real-time with the execution of the model by deploying and managing service instances. The resulting human-in-the-loop simulation of Cloud services can be used both for training purposes and for the (semi-)automated support for the real-time monitoring and management of the actual service instances and their computing resources.

Chapter 5 Cloud technology has become an invaluable tool to the IT business, because of its attractive economic model. Yet, from the programmers' perspective, the development of cloud applications remains a major challenge. In this paper we introduce a programming language that allows Cloud applications to monitor and control their own deployment. Our language originates from the Abstract Behavioral Specification (ABS) language: a high-level object-oriented language for modeling concurrent systems. We extend the ABS language with Deployment Components which abstract over Virtual Machines of the Cloud and which enable any ABS application to distribute itself among multiple Cloud-machines. ABS models are executed by transforming them to distributed-object Haskell code. As a result, we obtain a Cloud-aware programming language which supports a full development cycle including modeling, resource analysis and code generation.

This thesis is derived work from the publications:

- Bezirgiannis, N. and Boer, F. d. ABS: A High-Level Modeling Language for Cloud-Aware Programming In SOFSEM 2016
- Albert, E., Bezirgiannis, N., Boer, F. d., and Martin-Martin, E. A Formal, Resource Consumption-Preserving Translation of Actors to Haskell. In LOPSTR2016
- Azadbakht, K., Bezirgiannis, N., Boer, F. d., and Aliakbary, S. A High-level and Scalable Approach for Generating Scale-free Graphs Using Active Objects. In SAC2016
- Azadbakht, K., Bezirgiannis, N., and Boer, F. d. (2017a). Distributed Network Generation Based on Preferential Attachment in ABS. In SOFSEM2017
- Azadbakht, K., Bezirgiannis, N., and Boer, F. d. (2017b). On Futures for Streaming Data in ABS. In FORTE2017
- Bezirgiannis, N., Boer, F. d., and Gouw, S. d. Human-in-the-Loop Simulation of Cloud Services. In ESOCC2017.

The paper "Human-in-the-Loop Simulation of Cloud Services" was awarded the *Best Paper* of the Conference: (ESOCC) 6th European Conference on Service-Oriented and Cloud Computing.

Finally, all code developed during this thesis can be found at the git repository:

Chapter 3	[Bezirgiannis and Boer, 2016] [Albert et al., 2016] [Azadbakht et al., 2016]
Chapter 4	[Bezirgiannis et al., 2017]
Chapter 5	[Bezirgiannis and Boer, 2016] [Azadbakht et al., 2017a] [Azadbakht et al., 2017b]

Table 1.1: Contribution of publications to chapters of the thesis

`https://github.com/abstools/habs`

Note that the code was still in active development during the writing of this thesis; therefore, the latest implementation code might not reflect the hereby-included code snippets.

Chapter 2

Background: the ABS Language

The Abstract Behavioral Specification language[Johnsen et al., 2010a] (ABS for short) is a modeling language for concurrent systems. As such, it is well suited for describing, designing, and prototyping highly-concurrent computer software.

The ABS language is formally specified: the language's syntax and behaviour are not comprised merely of textual specifications or broader technical standards, but instead defined rigorously by means of mathematical methods. Since ABS is formally defined this makes it easier to analyze ABS models for possible deadlocks [Albert et al., 2014a, Albert et al., 2015b, Giachino et al., 2016b] or resource allocation [Albert et al., 2014a] and even fully verify properties over user-written functional specifications[Din et al., 2015]. Furthermore, the ABS formal semantics are laid out in a specific way that enforces the user to avoid certain problematic scenarios which arise during concurrent programming, such as race conditions and pointer aliasing.

ABS is executable — unlike other more "traditional" modelling languages — which means that any well-formed ABS model can be executed (evaluated) by a computer system. The ABS user can thus experiment and test any well-formed ABS model (e.g. by model-based test-case generation using symbolic execution [Albert et al., 2015c]) or even generate ABS code that can be integrated in production systems — currently there exist several ABS backends which generate production code partially or completely.

The syntax and programming feel resembles that of Java. In the rest of this section we introduce the basic elements and features of the ABS language in a manual-like style.

2.1 Data structures

All structures that hold data in ABS are immutable — with the exception of object structures, see section 2.3. An immutable structure cannot be updated in-place (mutated); instead the structure is copied into a new place in memory and its substructure updated. A common optimization is to not copy anew the whole updated structure but only its updated segment. Despite the obvious drawbacks of memory overhead and performance cost of copying, immutable data are considered beneficial in a concurrent but most specifically parallel programming setting for three reasons:

(a) Code can be written that does not have side-effects. This makes it easier for the user to *reason* about his/her program with the use of referential transparency (also known as equational reasoning) as well as the prover (human or not) to *analyse and verify* the code.

(b) Multiple threads can operate (i.e. read) the same location, but since the data does not change, the ordering in which different threads access it does not matter (no data races).

(c) The memory model becomes simpler; the compiler can thus apply code optimizations much more liberally.

The basic immutable data structures are the so-called *primitive data types* and consist of: `Int` standing for arbitrary-precision integers, `Rat` for arbitrary-precision rational numbers, `String` for (immutable) strings of Unicode characters. Integers can be implicitly converted to rationals (for more details, see section 2.4.2) but the other way around (downcasting) can only be done through explicit conversion (by using the function `truncate`), to avoid implicit (in other words, hidden) loss of precision errors in written ABS programs. All these primitive types are builtin inside ABS and cannot be redefined by the user or syntactically overwritten. Furthermore, there exist a special *builtin type* named `Fut<A>` which stands for a single containers of a value (of type `A`) that may be delivered sometime "in the future". For more about futures, see section 2.7. Since futures do not have a literal representation, they can be overridden. Example code of primitives and the special `Fut` is briefly given:

```
1               // Integer
1/1             // Rational
"text"          // String
obj!method();   // Futures created by async. method calls, see section 2.7
```

New user-written data structures can be given in the form of algebraic data types. Algebraic datatypes are high-level data structures defined as *products* and/or *sums* of types — types being other algebraic datatypes, primitives, and, in case of ABS, also of object types. A product groups together many data of different types, notated in set theory as $A * B * C \ldots * N$ where A,B,C,...,N are arbitrary types. Products resemble **struct**s in C-like languages and are denoted in ABS by:

```
data TypeName = ConstructorName(A,B,C,...,N);
```

where TypeName is the name of the type (required since ABS is statically-typed, see section 2.4) and ConstructorName is the name of the data constructor; in principle, a declaration of a data constructor name is not necessary unless the algebraic datatype contains also sums, but for the convenience of uniformity it is commonly required to give a constructor name for products as well. The most popular example of product types are *tuples*, with a triple of integers defined in ABS as:

```
data MyTriple = MyTriple(Int, Int , Int );
```

Sum types (also known as discriminated unions, tagged unions) groups together distinct types under a single "category" (type). The notation in set theory is $A + B + C + \ldots + N$ where $A, B, C, \ldots, N$ are arbitrary types (algebraic or object types) *as well as* product types. In other words, a sum type of $A, B, C, \ldots, N$ means that when a user "holds" a data structure with type $A + B + C + \ldots + N$, the contained value is of type either A, B, C, or N. The canonical example of a sum type is the boolean, given in set theory notation as $True + False$ and in ABS as:

```
data Bool = True
          | False;
```

where True, False are constructor names of their "nil-sized product types". The user could achieve the same in C-like languages with **enum** BOOL {false,true};. The extra power of algebraic datatypes shines when intermixing sums together with products; in set theory denoted by $(A*B)+(C)+(D*E*Z)+$... (parentheses added only for clarity, strictly speaking they are unnecessary since $*$ takes precedence over $+$) whereas in ABS language:

```
data Type = Constructor1(A,B)
          | Constructor2(C)
          | Constructor3(D,E,Z)
          |  ...;
```

Constructor names (e.g. Constructor (1,2,3) become important in sum types of statically-typed language since it allows us to *safely* (i.e. statically at compile-time) pattern-match on discrete values of possibly different, distinct types. Furthermore the contained types can be parametrically polymorphic with the use of type-variables:

```
data Either<TypeVar1,Typevar2> = Left(TypeVar1)
                               | Right(TypeVar2);
```

where TypeVar1 and TypeVar2 are type-variables standing for any possible type (algebraic or object type) which will be instantiated (be known) at use site.

The ABS language specification comes with a Standard Library that defines certain common algebraic datatypes such as Bool, Maybe<A> and List<A>, Set<A>, Map<A,B>:

```
export Bool, True, False;
export Maybe, Nothing, Just;
export List, Nil, Cons;
export Set,Map;
data Bool = True | False;
data Maybe<A> = Nothing | Just(A);
data List<A> = Nil | Cons(A,List<A>);
data Set<A> = //implementation;
data Map<A,B> = //implementation;
```

Note that Set<A> and Map<A> are so called *abstract* algebraic datatypes because their concrete implementation is not accessible outside of the module they are defined in (for our case inside ABS.StdLib). This is achieved by exporting only the types (i.e. Set, Map) and not the data constructors to the types, making them not accessible outside of the module. Abstract datatypes offer a two-fold advantage:

(1) operations on such datatypes preserve their invariants (e.g. no duplicate elements in a set or keys in a map, ordering, etc.) since the user cannot manipulate the data constructors of these types directly (by case-expression pattern-matching) but only through provided safe (in the sense of invariant-preserving) operations (functions).

(2) the individual (ABS) backends have the freedom to choose different purely (or not) functional datastructure implementations for those abstract datatypes.

2.2 Functional code

At its base, ABS adheres to a functional programming paradigm. This *functional layer* provides a declarative way to describe computation which abstracts from possible imperative implementations of data structures. Furthermore, ABS is said to be *purely* functional because inside any ABS program functional code cannot be mixed with side-effectful code (section 2.3). This pure/impure code distinction is achieved in ABS completely *syntactically*, compared to other purely functional languages where the same result is achieved at the type-system level (e.g. monads in Haskell).

At the centre of functional programming lies the *function* which is a similar abstraction to the *subroutines* of structural imperative programming, in the sense that it permits *code reuse*. However, unlike procedures, pure functions do not allow sequential composition (; commonly in C-style) since they completely lack side-effects. In the same manner, there is no need for an explicit **return** directive as the right-hand side evaluation of the function is the implicit return result (as it is mathematics). Note that sequential composition (;) is not the same as functional composition ($f \circ g$) because we are not composing right-hand side outputs of the functions but their underlying effects. The syntax of declaring an ABS function is:

```
def ResultType f<TyVar1,...TyVarN>(ArgType1 arg1, ..., ArgTypeN argN) = <expr>;
```

where f is the name of the function, arg1, ..., argN are the names of the formal parameters that the function takes (with their corresponding types) and ResultType is the overall type of the right-hand side expression. Furthermore, TyVar1, TyVar2, TyVarN are the typevariables that may appear inside formal parameters' types and/or ResultType. In this manner, functions can be parametrically-polymorphic, similar to to algebraic datatypes. Function definitions associate a name to a pure expression which is evaluated in the scope where the the expression's free variables are bound to the function's arguments. The functional layer supports pattern matching with a case-expression which matches a given expression against a list of branches.

An expression in ABS is either a primitive value (e.g. 1, 1/1, "text"), an applied data constructor (e.g. Left(3)), a fully applied function call (e.g.

f(True,4), ABS does not support partial application) an identifier (formal parameter or not), a case-expression, a *let-construct* or a combination of all of the above. A *let-construct* has the form let (Type ident) = <expr1> in <expr2> and binds the newly introduced identifier ident to point to expr1 inside the scope of expr2. The result-expression of a let-expression is the β-reduction of expr2 after capture-avoiding substitution of ident with expr1. The declared Type can be used to upcast the identifier if-and-only-if Type is a subtype of the expr1's actual type.

A case-expression is used to deconstruct a value of a datatype to its subcomponents and then assign particular identifiers to (some of) these subcomponents. This case analysis only makes sense for (non-abstract) algebraic datatypes, where the user has the ability to look inside the data constructors of the particular datatype. Other datatypes (primitives, abstract, algebraic, or object types) cannot be deconstructed and analyzed; only an identifier name can be assigned to them, similar to let-construct modulo the possible subtyping conversion. An example of the use of case-expression is given below:

```
def A fromMaybe<A>(A default, Maybe<A> input) = case input {
    Nothing => default;
    Just(x) => x;
    };
```

Each pattern => <expr> is a distinct branch of the case-expression. A (sub)-pattern can also be a wildcard (syntax: _) which matches any (sub-)component but does not bind it to an identifier. It should be mentioned that ABS does not do any case-pattern exhaustiveness search, which means that the ABS user can define partial functions, e.g.:

```
def A fromJust<A>(Maybe<A> input) = case input {
                        Just(x) => x;
                      };
```

which will throw a runtime exception (see section 3.2.1) when trying to evaluate fromJust(Nothing). Such data "accessors" are commonly used in functional languages, so the ABS language provides a shorthand for introducing such accessors (as syntactic sugar) at the point of the algebraic datatype declaration. For example, the above function will be implicitly defined, simply by annotating the constructor:

```
data Maybe<A> = Nothing
              | Just(A fromJust);
```

Finally, all primitive and algebraic data types provide default implementations for operations of (well-typed) structural equality (==) and lexicographical ordering (>,<,<=,>=).

2.3 Side-effectful and OO code

Keeping some form of state becomes handy when implementing certain algorithms, both for brevity and performance reasons. Stateful code also caters for C(++) and Java programmers, as the side-effectful and OO layers are much more familiar to them than the functional layer. ABS does not implement *stateful* computations through purely-functional abstractions (such as the State monad), but through the use of imperative programming (i.e. sequencing statements that possibly have side-effects). Furthermore, unlike an “observably-only” side-effect-free implementation of state (e.g. the ST monad of Haskell) ABS employs the full, *side-effectful* implementation of state as found in common imperative languages — in contrast, Haskell uses the IO monad. The reason that ABS uses side-effectful code is that albeit a modeling language, it allows certain observable communication with the real-world environment (e.g. println , readln, HTTP API) to facilitate user interaction during simulation (Chapter 4) or distributed computation (Chapter 5). As mentioned in section 2.2, ABS syntactically restricts the appearance of side-effectful code inside (purely) functional code. As such, side-effectful code can appear in ABS inside block scopes — a block is delimited by braces { } — i.e. the main-block (like the main procedure in C), every method-block (i.e. method body), while-block, if-then-else and if-then blocks.

The notion of local state in any imperative language is represented by *local variables.* Variables can be declared anywhere inside a method's body. The total scope of any variable is the scope from the start of its declaration line until the end of the current block. After declaration, they can appear inside expressions, be assigned and reassigned but not re-declared in the same or deeper scope. Furthermore, primitives (Int, Rat, String) and algebraic datatypes are forced to take an initial value, whereas object types and the special future type can be left uninitialized which will default them to null and unresolved future, respectively. An example of local variables inside a main block:

```
{ // main block can appear once per module

Int i = 3; // declaration/ initialization of a primitive
Maybe<Fut<String>> j = Nothing; // declaration/initialization of an ADT
i = i+1; // (re)assignment

Interf o; // declaration-only of an object type
Fut<String> f; // declaration-only of the special future type

j = Just(f); // (re)assignment

return Unit; // Unit returned by main, can be omitted

}
```

The main-block and every method-block can have a `return expr;` statement appearing strictly as the last statement of the block — this is too strict, since it would suffice to occur at every tail position so as to have a unique `return` point and no early exit of the method, but for clarity reasons the ABS language opted for a single-only return at the unique last position. If the `return expr;` statement is omitted, it defaults to `return Unit;` where unit is the singleton tuple (() in Haskell).

ABS is object-oriented: users can write classes which have a number of method definitions and fields. Fields can be declared in two positions:

```
class ClassName(<decl-pos1>) {
  <decls-pos2...>
  <method definitions...>
}
```

Fields at position-2 have the same initialization behaviour as local variables. Position-1 fields are instead left uninitialized and will be instead at creation time passed by the object creator as parameters (e.g. `new ClassName(params)`).

Fields can be referenced and reassigned inside any block with the prefix `this.fieldName`; fields have the same scope as their class. The special keyword `this` points to the currently executing object, much like Java. It is a syntax error for the main block to use the `this` or `this.fieldName` notation, since the main block lacks a `this`-object. An example of a class with one method block definition which adds to a field counter and returns the old counter value is given as:

```
class ClassName(Int counter) {
  {
      // init-block
  }
  Int addMethod(Int input) {
    Int oldCounter = this.counter;
    this.counter = this.counter + input;
    return oldCounter; // oldCounter is a local variable
  }
}
```

Instantiated fields and methods of an object are not visible by default outside its class scope. In practice this means that an object cannot access (read or modify) the fields of another object directly (all fields are private), but only through a method call, and any object can by-default only call its local methods (e.g. via calling this.m();). Calling a method of another object is achieved through explicitly exposed methods, which are bundled in interfaces. You can find more about interfaces and how they are used for (sub)typing in section 2.4.2.

Each class can have a single constructor, named the init-block. If omitted, it defaults to the empty-statement block. After the init-block finishes executing, the new object reference will be returned to the new-caller who can now resume execution with its next statement (i.e. new is a synchronous call). Also, after the init-block has finished, a Unit run() method will be implicitly asynchronously called for; this method is used for proactive concurrent objects. In Section 2.7 you can find more about synchronous/asynchronous calls and concurrency.

ABS lacks pointers and support for pointer-arithmetic. The evaluation strategy of ABS is strict, namely call-by-value semantics, much like Java where for primitive types, the value is passed, and for object types, the object-reference is passed as a value. ABS provides several common control-flow constructs: if−then−else branches and while-loops; there is no explicit **break**ing out of while loops. Any pure expression can be lifted to a side-effectful one. A case-statement, where case-branches associate to (side-effectful) statements, can be used instead of the similar but pure case-expression. Finally, ABS defines the equality operation (==) between objects to mean their referential equality; however, the ordering of (same-type) objects is left unspecified.

2.4 Type system

ABS is statically typed with a strong type system (strong referring to no implicit type conversion). The type system offers both System-F-like parametric polymorphism and nominal subtyping, commonly found in mainstream object-oriented languages.

2.4.1 Parametric Polymorphism

Parametric polymorphism appears in ABS in both datastructures and functions, e.g.:

```
data List<A> = Nil
       | Cons(A,List<A>);

def A head<A>(List<A> input) = case input {
  Cons(x,_) => x;
  };
```

The A above is a *type variable*, which means that it can take any concrete type at instantiation time.

Contrary to mainstream functional languages, the let-construct in ABS is non-recursive and parametrically monomorphic. Unfortunately, unlike other languages, there is no way to circumvent this monomorphism restriction, e.g. with an explicit type signature, since type variables in ABS can only be introduced either at data-structure or function definition and not in let definitions. For comparison, Haskell provides in addition to the explicit type signature approach, a language pragma to completely turn off the monomorphism restriction across the program modules.

Note that methods in ABS are parametrically monomorphic (compared to functions). Furthermore, there is no support for higher-rank parametric polymorphism since ABS lacks first-class functions to start with.

2.4.2 Subtype polymorphism

We saw in the previous section that the functions and algebraic datatypes (i.e. the functional core of ABS) are governed by a System-F-like type system: parametric polymorphism with no type inference. Instead, objects in ABS (imperative layer) are exclusively typed by interfaces. An interface, much like mainstream object-oriented languages, is a collection of method signatures. An example of an ABS interface is shown below:

```
interface InterfName1 {
  Int method1(List<Int> x);
}
```

A class is said that to implement an interface by writing:

```
class ClassName(params...) implements InterfName1, InterfName2... {
  Int method1(List<Int> x) { ... }
  ...
}
```

The ABS typechecker will make sure that the class implements every method belonging to the implements list of interfaces.

Unlike mainstream object-oriented languages, classes in ABS only serve as code implementations to interfaces and can not be used as types; as stated, in ABS an object variable is typed exclusively by an interface of its class, as in the example:

```
{
  InterfName1 object1 = new ClassName();
  object1.method1(Nil);

  InterfName2 object2 = new ClassName();
  ...
}
```

In the above example, object1 can be called only for the methods of its interface type InterfName1 and object2 only for InterfName2 accordingly.

Besides typing objects, the interface abstraction in many object-oriented languages serves also the purpose of nominal *subtype polymorphism* while ensuring strong encapsulation of implementation details. An interface type B is said to be a *subtype* of interface type A (denoted as $B <: A$) if it includes all the methods of A (and all of A's supertypes successively) and perhaps only adds new methods where their signatures do not interfere with any of the included methods (from the "supertype" interfaces). In ABS we have to explicitly declare that an interface is a subtype of another interface by using the extends directive, as shown in the following example:

```
interface InterfName2 extends InterfName1 {
  Bool method2(Int y);
}
```

In other words we explicitly "nominate" InterfName2 to be a subtype of InterfName1 (hence the term nominal subtyping), by inheriting all of the InterfName1 methods (i.e. method1) and extending it with method2. This is in contrast to *structural subtyping* where we do not nominate the subtype relations of the interfaces but the relations are derived from what methods the objects do implement (i.e. their structure). For example, under structural subtyping if an object o1 implements two methods m1 with type t1, m2 with type t2 and object o2 implements only m2 with type t2, then object o1's overall type is a subtype of o2's overall type, thus o1 can be safely upcasted to o2's type. The main benefit of structural subtyping is that it makes it possible to infer the overall types of the objects, but it comes with the drawback of accidental subtyping (upcasting), when there exist methods among objects with same signature but different "purpose". With nominal subtyping, accidental upcasting does not occur since the user provides explicitly the subtyping relation during interface declarations. An example follows of the (implicit) upcasting in ABS:

```
InterfName2 o = new ClassName();
o.method2(3);

InterfName1 o_ = o; // upcasting to super interface if InterfName2<:InterfName1
o_.method1(Nil);// can only call method1, method2 is not exposed through object o_
```

Note that, besides object types (typed by interface), the primitive types Int and Rat, albeit not represented through (mutable) objects, are associated by a subtype relation as well, where Int is a subtype of Rat, i.e. $Int <: Rat$.

2.4.3 Variance

Combining parametric polymorphism with (nominal) subtyping leads to the overall type system of ABS. Two important questions that arise in such a type system is a) what is the default *variance* of the abstractions offered by the language and b) is the user able to manually (as in syntactically) change their variance.

Generally, there are three different notions of variance:
Assuming B subtype-of A, i.e. $B <: A$,

(i) An abstraction C is *covariant* iff C<B> <: C<A>.

(ii) An abstraction C is *contravariant* iff C<A> <: C<B>.

(iii) An abstraction C is invariant if it cannot be further subtyped: neither (i) nor (ii) hold.

For certain abstractions, there are sensible variance defaults. E.g. immutable algebraic datatypes can be covariant by default, and pure functions are contravariant in their input types and covariant in their output type. There are reasons, however, that a user wants to change or restrict the default variance of an abstraction, e.g. a user wants to make an abstraction invariant because they know that the abstraction does not have to be later subtyped, or the implementation of the abstraction poses certain restrictions which deem it invariant.

The standard ABS type system [Johnsen et al., 2010a] (given as type rules of type theory) does not completely specify the default type variance (a). Furthermore, when ABS uses the term "subtyping" it refers to the common-notion of width subtyping and not that of depth subtyping[1] Taken from the specification of the ABS language:

> $T <: T$ is nominal and reflects the extension relation on interfaces. For simplicity we extend the subtype relation such that $C <: I$ if class C implements interface I; object identifiers are typed by their class and object references by their interface. We don't consider subtyping for data types or type variables.

So it is left to the particular ABS compilers to define their support for the variance of ABS abstractions. Many compilers (Maude-ABS, Erlang-ABS, HABS) provide sensible defaults of covariant subtyping for algebraic datatypes with only for width subtyping which is the default subtyping we described in this section (not depth subtyping). Finally, there is no current syntactic extension to the ABS language to provide means for manually changing the variance of user-written code.

2.4.4 Type Synonyms

Standard ABS provides language support for type synonyms. A type synonym of ABS is an "alias" assigning a (usually shorter, mnemonic) distinctive name to an algebraic datatype, object type, type synonym, or a combination of those. An example of a type synonym in ABS is shown below:

```
type CustomerDB = Map<CustomerId, List<Order>>;
type CustomerId = Int;
type Order = Pair<ProductName, Price>;
type ProductName = String;
type Price = Int;
```

[1]The term depth subtyping quite differs in meaning than the commonly found (width) subtyping. For a general description of what is depth subtyping, see `https://en.wikipedia.org/w/index.php?title=Subtyping&section=5#Width_and_depth_subtyping`

2.5 Module system

The ABS language includes an elaborate *module system*, inspired by that of Haskell. Modules can be specified in the same file or in separate files. Each module has at most one main block. The ABS user decides which main block will be the entrypoint of the program at the compilation step. Furthermore, by not exposing some or all data constructors of an algebraic datatype, the ABS user can designate the datatype to be *abstract*, i.e. it hides its concrete internal implementation. An example of the different constructs of the ABS module system follows:

```
module MyModule; // the beginning of a new module

export D,f,x; // exports specific identifiers
export * from M; // exports everything of imported module M
export *; // exports all local and imported identifiers

import M.ident; // imports identifier from module M as qualified
import ident from M; // imports identifier from module M unqualified
import * from M; // imports all exported identifiers of M unqualified
```

2.6 Metaprogramming with Deltas

Class inheritance, also known as code inheritance, is abolished in favour of code reuse via delta models [Clarke et al., 2010]. A delta can be thought of as a non-line-based patch (generated by Unix diff program) or better even, as a higher-level C macro. Unlike common preprocessors that check only for syntactic errors of the macros applied, deltas can also be checked for semantic errors, i.e. if certain delta applications are invalid. An example of delta meta-programming in ABS, taken from [Gouw et al., 2016], follows:

```
delta RequestSizeDelta(Int size); // name of the delta
uses FredhopperCloudServices; // which module to apply on
modifies class ServiceImpl { // modifies class
  adds List<Int> sizes = Nil; // adds field
  modifies Bool invoke(Int size) { // modified method
    sizes = Cons(size, sizes);
    return original(size); // uses original code
  }
}
```

Software product lines can be conveniently realized through feature and delta models (i.e. groups of features, and groups of deltas; deltas implement the features) [Clarke et al., 2010]. Specific software products can then be generated from the product line by selecting the desired features, as shown briefly below:

```
productline ProduceLine;
features Request, Customer;
delta CustomerDelta(Customer.customer) when Customer;
delta RequestSizeDelta(Request.size) when Request;

product RequestLatency (Request{size=5});
product OneCustomer (Customer{customer=1});

root Scaling {
  group [1..*] {
    Customer { Int customer in [1 .. 3]; },
    Request { Int size in [1 .. 100]; },
  }
}
```

2.7 Concurrency model

The foundation of ABS execution derives from the actor model [Hewitt et al., 1973]. The actor model is a model of concurrent computation where the primary unit of concurrency is the actor. An actor system is composed of (many) actors running concurrently and communicating to each other unidirectionally through messages. Unlike other well-known models for concurrent computation, the actor model arose "from a behavioural (procedural) basis as opposed to an axiomatic approach" for example that of Milner's π-calculus and Hoare's CSP.

Although the actor model is well-studied and discussed, there is no wide consensus on what the actor model consists of and what not. Furthermore, for practicality or implementation reasons, widely-used actor software deviates from the original Actor model specification. Arguably, the closest software implementation to the Actor model currently can be found in the Erlang programming language. For this reason, most of the following actor code examples are represented in Erlang's syntax. What follows is a rough list of the key properties found in the Actor model:

- Share-nothing philosophy where actors have private state and do not share memory with each other, but communicate only and explicitly by messages.

- Sending a message to another actor is *asynchronous*. The message will be put in the receiving actor's *mailbox*. To receive a message, the actor picks a message from its mailbox, an operation which is (usually) *synchronous*.

- After receiving a message, an actor has the choice either to stop executing, modify its private state, create new actors, send messages or decide (at runtime) to receive a different message (i.e. change dynamically its behaviour).

- There is no pre-defined ordering of message delivery: specifically, no local ordering that dictates that the messages of an actor arrive in the same order they were sent by that actor; nor is there a global ordering where sending actors can prioritize their own message over other actors.

- Actors are uniquely — across the whole actor system — addressable. An actor's address becomes known to other actors either upon actor creation or when an actor explicitly exposes its own address (commonly named **self**, which can be thought as OO's this):

```
% creates a new actor to run function(args). Returns the new actor's address
OtherActorId = spawn(function, [args]),
% sends its own actor address (self) to another actor as a message
OtherActorId ! (self, payload).
```

The concurrent execution model of ABS is the result of combining the object-oriented paradigm with the actor model. Specifically, on top of the synchronous method calls of (passive) objects of OO languages, ABS adds support for inherent concurrency and asynchronous communication between such objects: the result is called an *active object*.

The active object (also often named concurrent object) is based on the usual object found in mainstream OO languages, with object caller, object callee (this in ABS) and synchronous method call (callee.methodName(args)). Influenced by the actor model, the active object is extended with a mailbox for receiving messages. As in the actor model, there is no defined message arrival ordering inside the mailbox. Unlike the actor model, messages in ABS are not arbitrary (and possibly untyped) atoms, but instead type-checked methods that the callee explicitly exposes (via interfaces). Sending such a method (as a message) is accordingly named making an asynchronous method call (`callee !methodName(args)`).

```
object . method(args); // synchronous method call
object ! method(args); // asynchronous method call
```

A further deviation from the actor model is that the communication between ABS active objects is by default two-way whereas using the actor model we would need two (unidirectional) messages: a message with the request payload (plus the **self** identity) and a response message to "self" actor, plus the response payload. In active objects this is encapsulated inside the method's definition: the request payload are the method's actual parameters and the response payload is the return value.

```
main() ->
  Actor = spawn(className,[]),
  Actor ! {method,args,self}, % make an asynchronous method call
   ...
  receive
    Response -> doSomethingWithResponse(Response)
  end.

className() ->
 receive
   {method,Args,Sender} => Response = method(Args),
                           Sender ! response()
 end.

method(Args) = <impl>
```

```
{
  actor = new ClassName();
  actor ! method(args); // no need to send self (or this)
}

class ClassName() {
  ResponseType method(<args>) {
       ResponseType response = <impl>;
       return response; // the response is sent when return is called
  }
}
```

Another difference is that this two-way communication is a first-class citizen of the ABS language, called *future* and represented as `Fut<A>`. Upon establishing the communication, a future is created and assigned a unique identity among the active-object system. In the simple actor model (e.g. Erlang) a future abstraction has to be manually implemented by perhaps some unique tagging.

```
{
  actor = new Class();
  Fut<ResponseType> future1 = actor ! method(args); // asynchronous method call 1
  Fut<ResponseType> future2 = actor ! method(args); // asynchronous method call 2

  Bool b = future1 == future2; // FALSE identity comparison
   ...
  ResponseType response1 = future1.get; // block until response is ready
  doSomethingWithResponse(response1);
}
```

Get-blocking operation Holding a future is similar to holding a non-blocking reference to the "future" result value of an asynchronous method call. Instead, reading this future value (futureReference.get) is an operation which will block until the asynchronous method call has finished and the result has been communicated back. Futures are not restricted only to the caller but can be passed around and read from other objects; however, futures are written only once and only by the callee object. Futures can be tested in ABS for equality (==) based on their assigned identity (referential equality). The standard of ABS does not define a specific ordering on futures.

An ABS system is comprised of active objects executing their actions *concurrently* between each other, i.e. sending messages (method calls), receiving messages (method calls), sending responses (return), waiting for responses (get). As in the actor model, the level of concurrency (scheduling/interleaving) between active objects (actors) is left unspecified — although it usually assumes some starvation-freedom guarantees.

The ABS language adds an extra abstraction on top of active objects: the option of *grouping* active objects together. Every active object strictly belongs to one such group (named Concurrent Object Group, COG for short). To create an active object and put in a brand-new COG, the user uses the expression new, whereas to create an active object inside the current COG the expression new local:

```
InterfName object1 = new ClassName(params); // new object in a new COG
InterfName object2 = new local ClassName(params); // new object in current COG
```

In the simplest case where each active object lives in its own COG (through using only new), the same as before holds where the active objects in the system are executed concurrently (preemptively scheduled to avoid starvation). When forming larger groups (size > 1), each COG will essentially be a preemptively scheduled entity, whereas the active objects inside each COG will be *cooperatively scheduled* for concurrency. By "cooperation" we refer to the *intra-object* (i.e. inside the same COG) scheduling of ABS processes and not the synchronization between objects (inter-object communication), which is achieved through the previously describe get operation.

Cooperative scheduling means that an active object can decide to deliberately (syntactically) yield control to another object of the same COG. In other words, a COG can be seen as an individual (as in independent) processing unit where *at most one* active object is running on. Active objects on the same COG do not share their mailboxes, but share their processor (COG) for resources in cooperation. An active object "cooperates" by deciding to "pause" its execution and give the processing resources to any other object of the same COG. An active object may be later given back the resources to resume execution — same as a *semi-coroutine*, also called *generator*.

This cooperation (yield of control-execution-resources) manifests in ABS in three distinct forms:

Yielding unconditionally. The suspend statement releases control of the currently executing active object to some other object of the same COG (including possibly itself).

Awaiting on futures. A statement with the form await futureRef1? & ... & futureRefN?; means that the current object yields control and will not be resumed at least until all the given futures have their values ready. In contrast to futureRef.get; this does not block the whole execution unit (COG).

Awaiting on booleans. A statement of the form await booleanExpr1 & ... & booleanExprN means that the current object yields control and will not be resumed at least until all the boolean expressions evaluate together to True. Boolean awaiting makes only sense when used with boolean expressions that can change, i.e. contain some object fields, e.g. await this.x==this.y*2.

The arguments of the latter two forms can be combined with the operator (&), which basically means that the active object decides to yield at least until the specified futures have been resolved and the boolean conditions evaluate to True at the same time, e.g. `await fut1? & this.x>3 & fut2? & this.x==this.y*2`.

Since it is common practice to write code that includes an await on future following by its get, the ABS provides some syntactic sugar:

```
{
 A result = await o!m(args);
 // is syntactic sugar for
 Fut<A> hygienicRef = o!m(args);
 await hygienicRef?;
 A result = f.get;
}
```

It is worth mentioning that although the ABS standard leaves the preemptive as well as the cooperative scheduling underspecified, many ABS backends employ some concrete strategy, whereas ABS analysis and verification tools may explore many (if all) schedulability options.

Even with scheduling of messages being open to interpretation, the structure of the ABS language and its concurrency model avoid common problems that arise in concurrent programming, such as race conditions, deadlocks and pointer aliasing. First, the immutability of datastructures serves as the ground base to avoid a lot of race conditions that commonly arise in imperative programming. Moreover, futures which are commonly shared between objects (and their processors/threads) are also write-once (immutable). Secondly, the fields of any objects are not directly exposable to other objects, which leads to less incidents of pointer aliasing. Finally, the scheduling points in ABS are always explicit (i.e. suspend or await) which makes it easier to reason about the possible interleavings of an ABS program.

Note. The old ABS standard specified that synchronous method calls `caller.m(args)` where the caller and the callee belong to different COGs is a runtime error. The new version of ABS allows such synchronous method calls between different COGs, and translates them to the sequence `Fut<A> hygienicRef = caller!m(args); hygienicRef.get;`. This is not semantically equivalent to the synchronous method call of OO languages (also for same-COG objects), because although the caller blocks during this call, the callee does not “immediately” execute the corresponding method body; instead, the method is put in the mailbox to be later (undetermined and thus not immediate) executed.

2.8 History of ABS

The ABS language has its origins in the Creol language which is in turn the continuation of the SIMULA language. The Creol language [Johnsen et al., 2006] had features that the current (as of 2017) ABS standard has since abolished, e.g. cointerfaces (interface typing for callers as well) and class (code) inheritance (replaced instead with Delta metaprogramming). The current ABS enhanced the initial Creol language with support for algebraic datatypes with parametric polymorphism and pattern matching, pure expressions, and Concurrent Object Groups, inspired by the JCoBox Java extension [Schäfer and Poetzsch-Heffter, 2010].

The ABS language has been syntactically and semantically evolved through three successfully-completed European projects:

1. CREDO: "Modelling and analysis of evolutionary structures for distributed services". 2006–2009, `https://projects.cwi.nl/credo/indexpub.html`
2. HATS: "Highly Adaptable and Trustworthy Software using Formal Models". 2009–2013, `http://hats-project.eu`
3. ENVISAGE: "Engineering Virtualized Services". 2013–2016, `http://envisage-project.eu`

2.9 Comparison to other concurrent, modeling languages

The most well-known modeling language is the *Unified Modeling Language* (UML). Albeit a general-purpose modeling language, UML focuses on the design of software systems (mostly object-oriented-based), similar to ABS. Unlike ABS, UML is not in general executable, although there have been certain tools (Microsoft Visual Studio, Eclipse IDE) that can generate program code which derives from UML models. Moreover, UML is defined by standard committees (Object Management Group, International Organization for Standardization) and is not defined (as in the case of ABS) using rigorous formal-methods procedures. UML support for modeling concurrency (through means of interaction diagrams, e.g. sequence diagram, communication diagram, interaction overview diagram) is arguably not adequate to capture the interdependencies, evolution over time (creating/destroying actors) and inherent indeterminacy (unbounded non-determinism) of concurrent as well as distributed systems.

The Process Meta Language (PROMELA) of the SPIN model checker [Holzmann, 2003] is an executable modeling language used to verify (by means of model-checking) concurrent software systems. The concurrency unit in PROMELA is the process; the execution of processes is interleaved (unless using atomic blocks) and the communication between processes is achieved through (buffered) channels which can be globally shared. Also PROMELA has a "choice" construct to express bounded non-determinism. In those respects, PROMELA's model of computation resembles more that of Communicating Sequential Processes (CSP), than the actor model. Furthermore, the language has, justifiably, limited support for complex data types, since "larger" datatypes is one of the culprits for the problem of state explosion during model checking.

Rebeca ([Sirjani et al., 2004]) is a verifiable modeling language which combines as well the object-oriented paradigm with the actor model. If the number of "active objects" and size of their mailboxes are bounded, Rebeca can generate models in lower-level code so as to apply model checking by using external model checkers, e.g. SPIN. Rebeca also encapsulates the message passing of the Actor model behind the (asynchronous) method calls, but the communication is one-way, i.e. such asynchronous methods cannot return values. Since there are no such "responses" to asynchronous method calls, there is also no need for futures and their awaiting mechanism. In other words, the active objects of Rebeca run in an interleaving manner between them, but their methods only "appear" to run atomically, meaning that each method is executed from start to end; there exist no mechanism (suspend, await) to jump midway the method's body to a different method execution of the same object. Finally, there is no support for algebraic datatypes and COGs, but there is syntax for representing bounded non-determinism (choice).

Eiffel (`http://eiffel.org`) is one of the first statically-typed object-oriented languages, and became known for its design by contract philosophy, i.e. associating invariants to classes and contracts in the form of pre- and post-conditions around method bodies. Unlike ABS, these conditions will not be checked statically, by analysis and verification tools, but only at runtime (like the familiar assertions). The language provides limited support for concurrency, mainly by means of traditional threading, but like ABS offers private-only fields and limited form of parametric polymorphism. In a perhaps similar spirit, the Java Modeling Language (JML) tries to introduce a lightweight method of design-by-contract and formal verification to (existing) Java object-oriented codebases.

A more detailed discussion of real-world concurrent programming languages and runtimes can be found in section 3.9.

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:

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

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 type-checking 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 $newIORef :: a- > IO(IORef a)$, 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 $readIORef :: IORef a- > IO a$ or changed by calling $writeIORef :: IORef a- > a- > IO()$. 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 (co-variant) 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 `l2` with `up`, since the `up`cast 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-

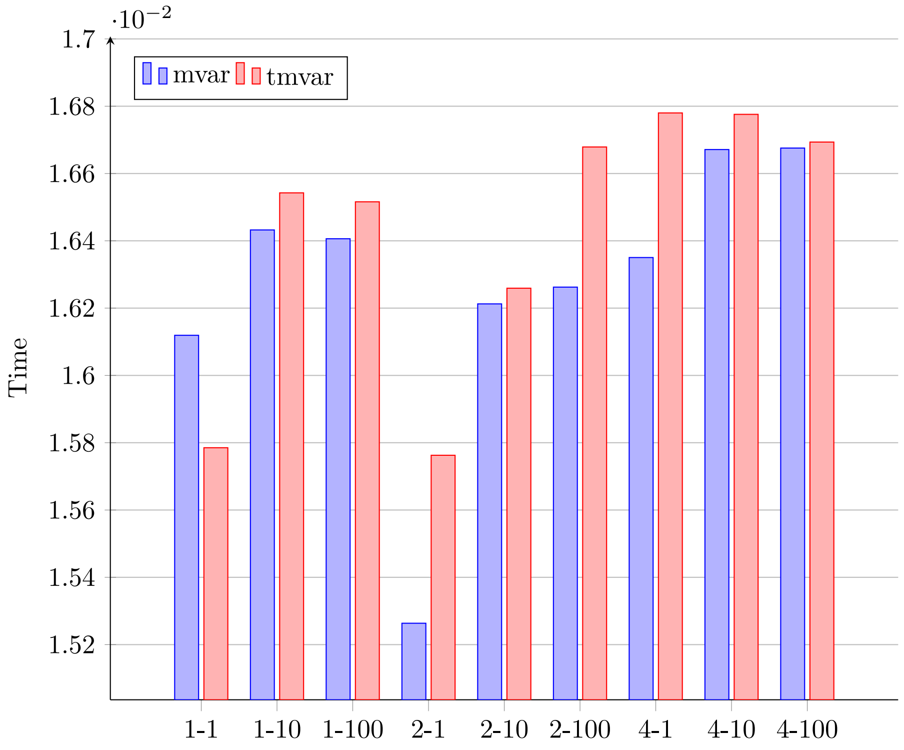


Figure 3.1: Implementing futures using MVar or TMVar on varying scenarios (workers-listeners).

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.

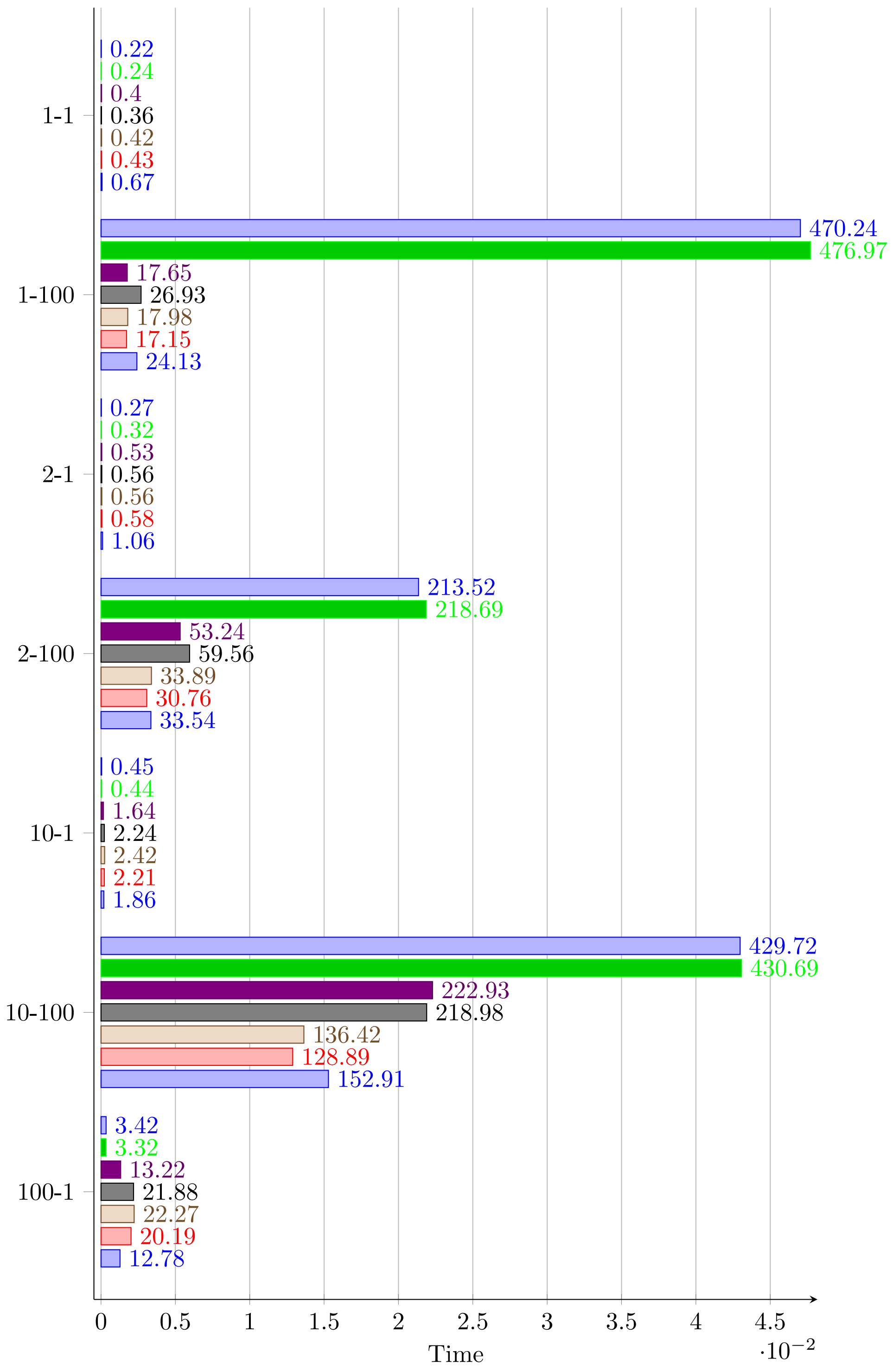
Java-ABS The Java-ABS backend was the first backend specifically developed to implement the Concurrent Object Groups (COGs) and has been superseded by the Erlang-ABS backend.

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.

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

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

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

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

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

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

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

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

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

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

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

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 $\Sigma = IVar \to Int$ 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 $\tau \in LVar \to Int$, 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) $h_1 : \mathit{Int} \rightarrow \Sigma$ and $h_2 : \mathit{Int} \rightarrow \mathit{Int}_\perp$, where $\mathit{Int}_\perp = \mathit{Int} \cup \{\perp\}$ ($\perp$ 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 $h(\mathit{count})$, and write $h(n)(x)$, instead of $h_1(n)(x)$, and $h(n)$, instead of $h_2(n)$. We will use the notation $h[\mathit{count} \mapsto n]$ 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 `getVal`(Σ, 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):

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

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

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., $\langle n : Q, h\rangle \to^n_S \langle n : Q', h'\rangle$ and $\langle C, h\rangle \to^n_S \langle C', h'\rangle$.

We remark that the operational semantics shown in Fig. 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→ 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→ 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→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→a is rewritten to take its current continuation as an extra last argument, as in f ':: args→(a→r)→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→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≡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 ::() →Stm and further to the "nullary" function k :: Stm. Accordingly the CPS type of our methods (functions) and statements (constructors) becomes f ':: args→Stm→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 → 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
```

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

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

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 `awaiting` 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 $\mathit{nextObject}(h, [\overline{o_m}])$, 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.

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

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

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

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

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

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

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

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

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

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 (Return$_A$), where the object list can increase or decrease one object, we use the following auxiliary functions. $\mathit{newQ}_{add}([\overline{o_m}], o_n, o_y)$ inserts the object o_y into $[\overline{o_m}]$ if it is new (i.e., it does not appear in $[\overline{o_m}]$), and $\mathit{newQ}_{del}([\overline{o_m}], o_n, q_n)$ removes the object o_n from $[\overline{o_m}]$

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

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. $\to$. 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 ${}^{c}[\![(h_1,s_1)]\!]^{-1} \to^{o_1}_{res_1} {}^{c}[\![(h_2,s_2)]\!]_c^{-1} \to^{o_2}_{res_2} \ldots \to^{o_{n-1}}_{res_{n-1}} {}^{c}[\![(h_n,s_n)]\!]^{-1}$.*

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 $\mathcal{M}(new) = \mathtt{c}$, 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*

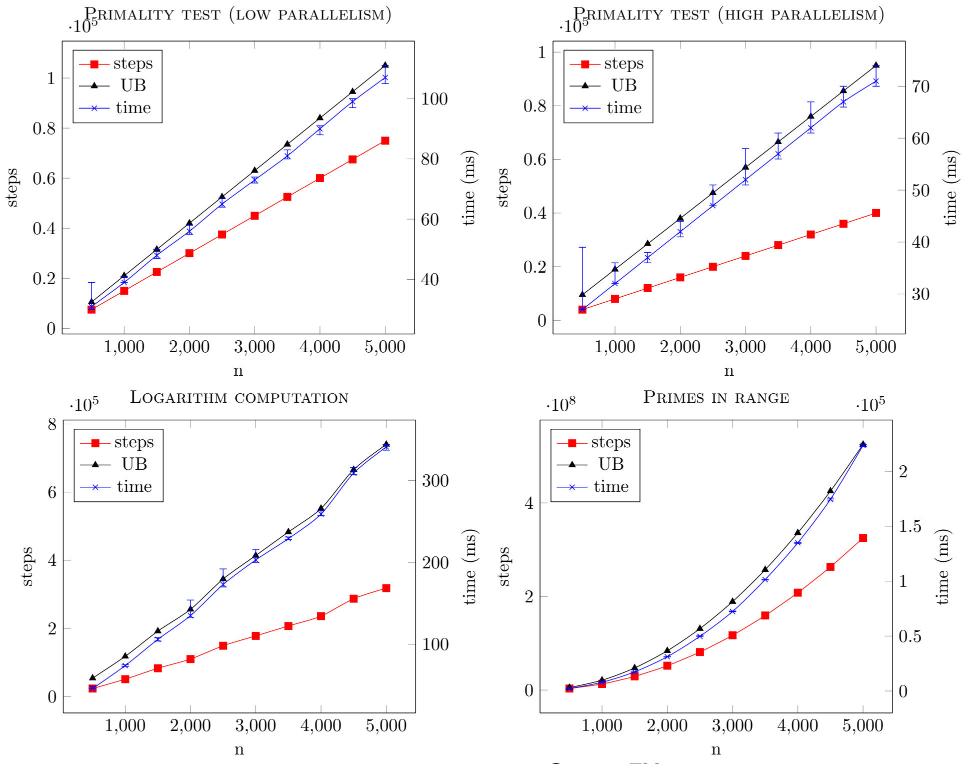


Figure 3.9: Execution steps vs. time (Intel® Core™ i7-4790 at 3.60GHz, 16 GB).

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 particular 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. $\to$. The completeness states that any $\to$-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 $\to$-step from the translated configuration.

Lemma 3 (Completeness of $\rightarrowtail$)**.** *If $A \to_S^{o_n} B$ 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)**.

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

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:

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

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 $[\overline{o_{n+1\to m}}] : [\overline{o_{1\to n}}]$, 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)**.

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

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:

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

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)**.

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

where

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

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:

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

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

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

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

- **(Internal)+(Sync)**.

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

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

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

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

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

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

- **(Internal)+(Return$_A$)**.

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

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

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

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 on h = (res,l,h')` *such that* $[\overline{o_k}] \equiv \mathit{updL}([\overline{o_m}], o_n, l)$*.*

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 $[\overline{o_m}]$.

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

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

- **(Assign)**.

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

If $k = $ `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:

```
Assign lhs (Val x) k' -> do
  (attrs `V.write` lhs) =<< (getVal x)
  updateObj $ Left k'
  return (res,
          [this],
          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)**.

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

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)**.

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

  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)**.

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

  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)**.

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

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$_S$).**

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

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 $\langle C, h\rangle$:

- $\mathit{objs}(C)$: returns the set of object identifiers in the set C.
- $\mathit{futs}(h)$: 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).

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

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

$$\begin{array}{rcl}
\|\epsilon\|_s & = & \epsilon \\
\|\mathtt{x:=}V;S\|_s & = & \mathtt{x:=}\|V\|_v;\|S\|_s \\
\|\mathtt{x:=new};S\|_s & = & \mathtt{x:=new};\|S\|_s \\
\|\mathtt{x:=f.get};S\|_s & = & \mathtt{x:=f.get};\|S\|_s \\
\|\mathtt{f:=x!p(\bar{z})};S\|_s & = & \mathtt{call(m,p(x,\bar{z},f))};\|S\|_s \\
\|\mathtt{f:=p(\bar{z})};S\|_s & = & \mathtt{call(b,p(this,\bar{z},\_))};\|S\|_s \\
\|\mathtt{await\ f};S\|_s & = & \mathtt{await\ f};\|S\|_s \\
\|\mathtt{return\ x};S\|_s & = & \mathtt{return\ x};\|S\|_s
\end{array}$$

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 $S_i \equiv res_i$)

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

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

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

By Lemma 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

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

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 $\langle C, h\rangle \rightarrow^n_b \langle C', h'\rangle$ then:*

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

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

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

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

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 $\leadsto$-step using rule (1) in [Albert et al., 2015b], reaching $\|B\|$:

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

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)**.

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

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 $\leadsto$-step using rule (3) in [Albert et al., 2015b], reaching $\|B\|$:

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

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)**.

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

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)**.

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

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.

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

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]:

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

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:

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

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

- **(Internal)+(Sync)**.

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

The translation of S_1 is

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

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 $\|S_2\|$:

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

$\|D\|^x_{sync}$ 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$_A$)**.

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

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]:

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

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`

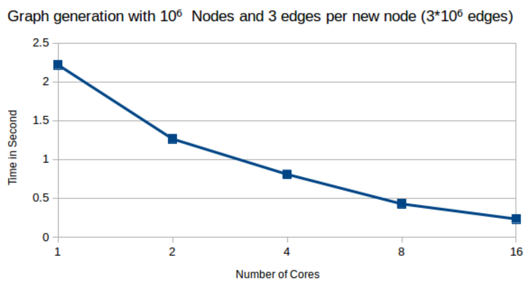


Figure 3.10

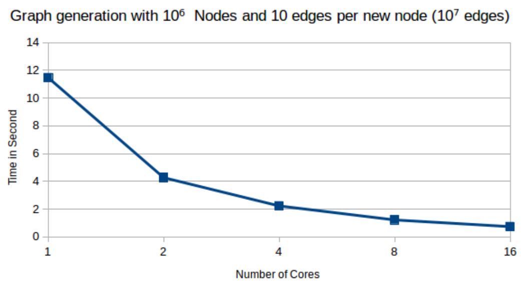


Figure 3.11

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

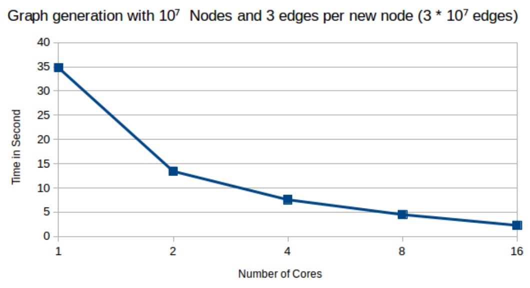


Figure 3.12

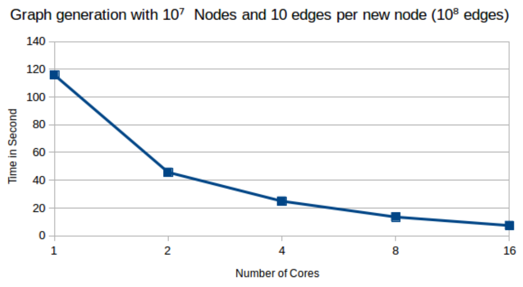


Figure 3.13

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 co-operative 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.

Chapter 4

Resource-aware Modeling in HABS

The standard ABS language, described in chapter 2, is adequate to represent models of concurrent object-oriented programs; the ABS user can make use of the ABS tool-suite to analyze, experiment, and execute such models. It becomes, however, more difficult for the user to express models which change their behaviour over *time*; such models are usually constructed during a simulation phase. The word *simulation* can take a broad meaning; here, we use the word to refer specifically to *computer simulation*: the (inexact) reproduction of a real-life process or system, performed with the aid of a computer. We implemented the timed extension of ABS with a real-time interpretation inside the HABS framework.

Furthermore, we model virtualized systems (named Deployment Components) directly inside ABS as first-class citizens of the language in section 4.3, as well as their virtualized resources (speed, memory, bandwidth) in section 4.3. At the end, we evaluate this extension to HABS in an industrial case-study by modeling and simulating real-world cloud environments.

4.1 Modeling time

[Bjørk et al., 2013] address the issue of time-varying models and simulation in ABS with a small extension of the language to deal with time; the entity *time* in their case is left abstract to accommodate all possible scenarios with different notions of time (symbolic or real-time) or units of time (seconds, milliseconds, days, etc.). The following ABS snippet encompasses the wholes new syntax of this “timed” ABS extension by means of an example:

```
{
  Rat i = 3.1;
  duration(i,i+1);
  await duration(i+1,i+2);
  Time n = now();
}
```

A duration(i,j); statement blocks the currently-executing Concurrent Object Group (COG) and all of its processes for less than j time and for the best case i amount of time; in other words, the blocked time is sampled from the interval $[i, j)$. The statement await duration(i,j); will instead block only the currently-executing ABS process for that amount of sampled time; the other processes of the COG can still be scheduled for execution in the meantime. Finally, there is the effectful expression now() which returns the current clock of the simulation in the abstract algebraic-datatype $Time$; this expression is used mostly for printing & debugging purposes. It is worth mentioning the fact that the rest statements of ABS do not "take" time — in the sense of abstract ABS time, they can still take perceived clock time — and treated by the timed extension of ABS as instantaneous.

For our case, we implement the Timed-ABS language extension as an extension of the HABS compiler & runtime, accordingly. We deviate from the initial work on Timed ABS ([Bjørk et al., 2013]) by providing a specific notion of time, that of the passage of real-world time — in short, real-time. This choice becomes important later on since it allows us to have *live* simulations where the human can interact with the computer's simulation, instead of having 'as-fast-as-possible" simulations. Another reason for implementing the Timed-ABS extension for HABS is that in the subsequent Chapter 5 that details the (cloud) distributed-computing part of HABS, the importance of time becomes more apparent in such a real-world setting, where the network latency of communication plays and workflows of cloud services dominate the structure of the model.

A different interpretation of time for Timed-ABS is that of symbolic time, which is is implemented in the Erlang-ABS backend. Specifically, the Erlang-ABS backend of ABS provides a symbolic interpretation of the abstractions modeling (CPU) time, that is, time is modeled by a symbolic clock which is advanced by the execution of a certain kind of statements, so-called duration statements. In contrast, in this thesis we introduce the new Haskell backend for ABS denoted by HABS, which is based on a source-to-source translation of ABS into Haskell and which directly relates the ABS abstractions of time to the underlying hardware clock. It should be noted that the term "real-time ABS" has also been used, for example in [Johnsen et al., 2012], to refer to the ABS abstractions modeling (CPU) time themselves. In this section, however, we use the term "real-time" to refer to the implementation of these abstractions with respect to some external clock, e.g., the hardware clock. This implementation allows for a different kind of simulation, so-called human-in-the-loop simulation, abbreviated in the sequel by HITL. In general this kind of simulations require human interaction and are used for training purposes. A typical example is that of flight simulations where trainees interact in real-time with a model of a plane in flight. Clearly, for such training to be effective the human interactions should be processed by the model in real-time as measured by the hardware clock.

4.2 Modeling virtualized hardware resources

Systems in ABS are composed of resources. Example of resources are the number of CPU cores, their speed, the total memory of the system, the network bandwidth, etc.. In this section we discuss how computing resources are modelled in ABS.

High-level annotations of the ABS code are used to specify the resource consumptions of the annotated statement ([Johnsen et al., 2012, Albert et al., 2014b]). For example to signal the overall-CPU resource consumption of a statement, we annotate it by `[Cost: intExp()] stmt`; which means in practice that *stmt* will be only completed (and its side-effects instantaneously realised) after some time where *intExp* amount of resource *Speed* has been provided and consumed by the currently executing deployment component. This model of deployment as executable ABS allows for a formal analysis of the constraints induced by the shared resources in terms of a formal cost model and its relation to a formalization of Service Level Agreements (SLA 's) as a property of a service metric function.

Whereas the *Cost* annotation induces the passage of time *locally* inside an abstraction of a system, a so called deployment component (see section 4.3), the timed-ABS extension of the language enables time to pass globally (over the whole model) always with respect to an external clock. The statement `await duration(min,max)` means that the current process will be rescheduled for execution only after *min* and less than *max* time steps from now have passed on the clock; the statement `duration (min,max)` will accordingly block the object and all of its process for that time. If the ABS clock refers to symbolic (abstract) time — used for synchronizing distinct parts of the model — then the models' execution is essentially a computer simulation; however, a model running on the real (hardware) clock defines a user-interactive simulation.

4.3 Modeling systems

We extend the ABS language with syntactic and library support for Deployment Components. A *Deployment Component* (DC), first described in [Johnsen et al., 2010b], is "an abstraction from the number and speed of the physical processors available to the underlying ABS program by a notion of concurrent resource". Over time, in ABS a DC has further evolved to include other virtualized resources of a computer system seen in the previous section, like CPU time, memory, and bandwidth, which allows to model virtual machines and in general other technologies, e.g. Docker containers, unikernels. We want to be able to deploy and execute objects on a Deployment Component and that requires at least the presence of CPU resources. In this section we only deal with "simulated" Deployment Components, i.e. machines that do not have actual computing resources but instead simulated ones, for tracking and predicting the possible utilization of real machines.

To be able to programmatically (at will) create and delete machines in any language would require modeling them as first-class citizens of that language. As such, we introduce DCs as first-class citizens to the already-existing language of ABS in the least-intrusive way: by modeling them as objects. Since Deployment Components are expressed by concurrent objects themselves they become an integral part of any ABS model. All created DC objects are typed by the interface `DC`. The minimal interface for deployment components contains the methods `shutdown` for shutting down and releasing the cloud resources of a virtual machine, and `load` for probing its average *system load*, i.e. a metric for how busy the underlying computing-power stays in a period of time. We use the Unix-style convention of returning 3 average values of 1, 5 and 15 minutes. After calling `shutdown()`, the DC object will point to `null`. The DC interface resides in the augmented standard library:

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

Similar to `this` identifier, a method context contains the `thisDC` read-only variable (with type `DC`) that points to the machine host of the currently executing object. A running ABS node can thus control itself (or any other nodes), by getting its system load or shutting down its own machine. However, after its creation, a running ABS node will remain effectively "idle" until some objects are created/assigned to it. The `DC` annotation can be used in conjunction with the `new` keyword to specify in which (possibly remote) DC the newly created objects which "live" and run:

```
[DC: dc1] Interf1 o1 = new Cls1(args..);
o1 ! method1(args..);
this.method2(o1);
```

Such objects dynamically deployed onto deployment components are named *remote objects* and share their resources. The `DC` annotation does not change the behaviour of the `new` keyword: it still creates a new object (inside a new COG), initializes it, and optionally calls its run method. Indeed, the unannotated expression `new Cls1(params)` is equivalent (as in syntactic sugar) to the annotated `[DC: thisDC] new Cls1(params)`. References to remote objects are indistinguishable to local object references and can be normally passed around or called for their methods. The ABS language specification and its cloud extension do not dictate a particular Garbage Collection policy — a specific implementation is provided for distributed HABS at section 5.1.3, but we assume that holding a reference to a remote object or future means that the object is alive, if its DC is alive as well.

Usually the ABS user does not create deployment components directly (i.e. by calling new DC), but instead through a higher object abstraction named *Cloud-Provider*, which serves both as a factory of deployment components as well as a communication endpoint (in the real and not simulated world this corresponds to the infrastructure service, e.g. Amazon AWS, OpenStack, Azure):

```
CloudProvider cp = new CloudProvider(params);
this . addInstanceDescription ( Pair(" c4_2xlarge_eu ",  map(Cons(Pair(CostPerInterval ,419),
  Cons(Pair(Cores,  8),  Cons(Pair(Memory, 1500), Cons(Pair(Speed, 31),  Nil )))))));
this . addInstanceDescription ( Pair("m4_large_eu",  map(Cons(Pair(CostPerInterval,  120),
  Cons(Pair(Cores,  2),  Cons(Pair(Memory, 800), Cons(Pair(Speed, 6),  Nil )))))  ));

DeploymentComponent vm1 = cp.launcInstanceNamed("m4_large_eu");
[DC: vm1] new WebServer(8080); // deployed object
```

4.4 A real-time implementation

In this section we introduce the ABS_RT Haskell backend of ABS and present its use by Cloud engineers so that they can interact in real-time with the execution of the model of the services offered on the Cloud. This interaction consists of deploying and managing service instances and allows Cloud engineers to acquire knowledge of the real-time consequences of their decisions. We illustrate this use of HITL simulation of Cloud services by an industrial case study based on the Fredhopper Cloud Services.

We augment the original HABS backend with support for the timed-ABS language extension, and name the resulting backend ABS_RT. The clock that ABS_RT uses is the available real-time hardware clock underneath. This means that compared to the backends with a symbolic clock (Erlang-ABS, Maude-ABS), the passage of time is not influenced by timed-ABS calls but instead by the real clock itself. The duration statement is implemented as a *sleep* call on the concurrent object's thread, whereas the await duration creates a new extra lightweight thread which will reschedule its continuation back to the original object thread after the specified time. The [Cost: x] annotations are translated to a executeCost() method call on the deployment component object as seen in Listing 4.1. The instrPS field refers to the number of instructions the particular deployment component is able to execute per second. The unit of time (default is seconds) is tunable as a runtime option.

```
Unit executeCost(Int  cost) {
    Int  remaining = cost;
    while  (remaining > this.instrPS) {
      duration (1,1);
      suspend;
      remaining = remaining − this.instrPS;
    }
    Rat last = remaining / this.instrPS;
    duration( last , last );
  }
```

Listing 4.1: The implementation of Cost annotation for the ABS_RT backend

It is worth nothing that the GHC runtime scheduler dictates that any "sleeping" thread will be re-activated (preempted) no sooner than the specified time, but may be later than prescribed (not precise). This does affect the reproducibility, in addition to the fact that there is no notion of simultaneous method calls (no specific ordering, thus non-deterministic hardware-dependent process-enqueuing of simultaneous callers) as it can be done with total ordering of symbolic time. Finally, we would like to mention that this real-time implementation as shown in Listing 4.1 is generic for any ABS backend that uses the hardware clock and implements *duration/await duration* as a *sleep()* system call. Indeed, it would be straightforward to port it to the Erlang-ABS and Java-ABS backends as well.

4.4.1 Comparison with symbolic-time execution

As briefly discussed in section 4.1, the Erlang-ABS backend also implements the Timed-ABS extension but with a symbolic clock as notion of time. The Erlang manual of ABS (at `http://docs.abs-models.org`) says that:

> Time only advances when all processes are blocked or suspended and no process is ready to run. This means that for time to advance, all processes are in one of the following states: the process is awaiting for a guard that is not enabled, the process is blocked on a future that is not available the process is suspended waiting for time to advance, the process is waiting for some resources, In practice this means that all processes run as long as there is work to be done.

At implementation side, the Erlang-ABS backend will execute all processes that are enabled in the current clock to completion, and will advance the time only if all of the processes of the system are blocked (as in idling). Then, the Erlang-ABS runtime will advance the clock to the smallest amount of time of a `duration` or `await duration` ABS statement.

The described above Erlang-ABS execution resembles that of timed automata, for example as is done in the model checker UPPAAL. There are certain reproducibility problems attached to this execution method. First of all, there exist the problem of "granularity of concurrency": the Cost resources although being rational numbers, are always distributed to processes of a COG with a granularity of 1 unit. For example, in a hypothetical situation of a DC with `Speed=3`, one process may "consume" 2 cost resources, while the other process can only "grab" 1 cost resource. A better approach would be to distribute the resources evenly to all the processes (for the example 1.5 to each process). This is currently hardcoded in the Erlang-ABS runtime and is not parameterizable. A further problem with reproducibility is that the Erlang-ABS runtime does not provide any "local" method ordering; the scheduled processes of a COG do not follow a queue pattern, where a process that arrived earlier will execute also earlier (FIFO), instead the processes are picked up for execution in arbitrary order. In other words, the scheduling policy of Erlang-ABS is non-deterministic. This leads to the inherent problem of non-reproducibility for certain ABS models running with the Erlang-ABS backend. Consider the artificial example of spawning a number of asynchronous methods:

```
module Test;

class C {
  Unit run() {
     Int i = 0;
     while (i<10) {
        this!m(i);
        i=i+1;
     }
   }
  Unit m(Int n) {
     println(toString(n));
  }
}

{
  new C();
}
```

The output of the above model's execution is non-deterministic with the Erlang-ABS runtime, varying between successive runs, e.g. `0 2 3 4 6 8 7 5 1 9` and `4 9 7 2 1 0 6 5 3 8`.

However, even with assumption of method ordering inside the COG the execution of an ABS model remains non-deterministic (thus non-reproducible simulation) since there is no fixed scheduler for which COG will execute next. In fact, certain runtimes (Erlang-ABS, HABS) execute the COGs simultaneously for the benefit of parallelism. Consider the following example of i number of COGs:

```
module Test;

interface R {
  Unit m(Int n);
}
class R implements R{
   Unit m(Int n) {
     println (toString(n));
  }
}

class S(R r, Int i) {
  Unit run() { r!m(i); }
}

{
  R r = new R();
  Int i=0;
  while (i<10) {
    new S(r, i);
    i=i+1;
  }
}
```

The output of the above example will again vary on successive runs. This leads us to consider for future work a simulation of Timed-ABS models where the execution is driven by a discrete-event simulation (DES) engine. In this way, we could achieve reproducibility since every event will be marked with its timestamp and all events are executed in total order. Another theoretical benefit is that such discrete-event simulations can be executed in parallel or distributed over different computers which may improve the execution performance compared to the real-time approach (HABS) as well as that of timed automata (Erlang-ABS).

4.5 Case study: DevOps-in-the-Loop

In this section, we evaluate the ABS_RT backend on an industrial case study. We integrated ABS_RT in a new tool-suite for human-in-the-loop simulations for cloud engineers. Other tools in the suite include the SAGA tool [Boer and Gouw, 2014] for the declarative specification of service metric functions, and SmartDeployer [Gouw et al., 2016] for the formalization of deployment requirements and the automatic generation of provisioning scripts. At the core of this suite is a new Haskell backend ABS_RT of the ABS modeling language which supports a real-time interpretation of the timing constructs of ABS. We further illustrate the use of our tool-suite by an industrial case study based on the Fredhopper Cloud Services. The underlying ABS model of the Fredhopper Cloud Services builds on the one presented in [Gouw et al., 2016] which focuses on automated generation of deployment actions. Here we extend that model to support HITL simulation and for the generation of more realistic deployment recommendations.

The general methodology underlying the use of ABS_RT in the HITL simulation of Cloud services involves the formalization of Service Level Agreements (SLA 's) as a property of a service metric function, as described in [Giachino et al., 2016a], with a new framework in ABS which captures various monitoring concepts – from QoS and SLAs to lower-level metrics, metric policies, and listenable and billable events. The monitoring framework allows the formal development and analysis of monitors as executable ABS.

Fredhopper[1] provides the Fredhopper Cloud Services to offer search and targeting facilities on a large product database to e-Commerce companies as services (SaaS) over the cloud computing infrastructure (IaaS). Fredhopper Cloud Services drives over 350 global retailers with more than 16 billion in online sales every year. A customer (service consumer) of Fredhopper is a web shop, and an end user is a visitor to the web shop.

The services offered by Fredhopper are exposed at endpoints. In practice, these services are implemented to be RESTful and accept connections over HTTP. Software services are deployed as *service instances*. The advantages of offering software as a service on the cloud over on-premise deployment include the following: to increase fault tolerance; to handle dynamic throughputs; to provide seamless service update; to increase service testability; and to improve the management of infrastructure. To fully utilize the cloud computing paradigm, software must be designed to be *horizontally* scalable[2]. Typically, software services are deployed as *service instances*. Each instance offers the same service and is exposed via the Load Balancing Service, which in turn offers a service endpoint (Fig. 4.1). Requests through the endpoint are then distributed over the instances.

The number of requests can vary greatly over time, and typically depends on several factors. For instance, the time of the day in the time zone where most of the end users are located, plays an important role. Typical lows in demand are observed daily between two am and five am. In the event of varying throughput, a different number of instances may be deployed and be exposed through the same endpoint. Moreover, at any time, if an instance stops accepting requests, a new instance may be deployed in place.

Architecture of the Fredhopper Cloud Services

Each service instance offers the same service and is exposed via Load Balancer endpoints that distribute requests over the service instances. Fig. 4.1 shows a block diagram of the Fredhopper Cloud Services.

Load Balancing Service The Load Balancing Service is responsible for distributing requests from service endpoints to their corresponding instances. Cur-

[1]`https://www.fredhopper.com/`
[2]`en.wikipedia.org/wiki/Scalability#Horizontal_and_vertical_scaling`

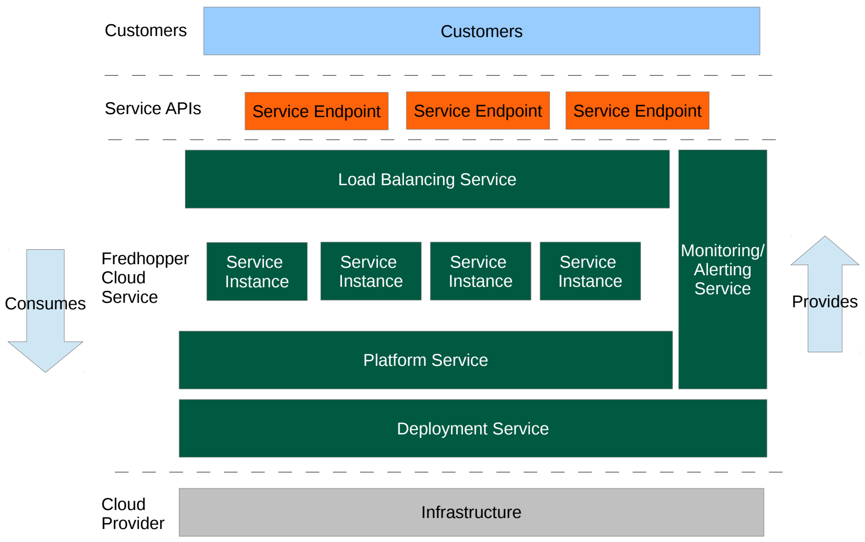


Figure 4.1: The architecture of the Fredhopper Cloud Services

rently at Fredhopper, this service is implemented by HAProxy (`www.haproxy.org`), a TCP/HTTP load balancer.

Platform Service The Platform Service provides an interface to the Cloud Engineers to manage customer information, deploy and manage service instances associated to the customers, and associate service instance to endpoints (load balancers). The Platform Service takes a service specification, which includes a *resource configuration* for the service, and creates and deploys the specified service. A service specification from a customer determines which type of service is being offered, the number of service instances to be deployed initially for that customer, and the kinds of *virtualized resources* on which the service instances should be deployed.

Deployment Service The Deployment Service provides an API to the Platform Service to deploy service instances (using a dedicated Deployment Agent) onto specified virtualized resources provided by the *Infrastructure Service*. The API also offers operations to control the life-cycle of the deployed service instances. The Deployment Service allows the Fredhopper Cloud Services to be independent of the specific infrastructure that underlies the service instances.

Infrastructure Service The Infrastructure Service offers an API to the Deployment Service to acquire and release virtualized resources. At the time of writing the Fredhopper Cloud Services utilizes virtualized resources from the Amazon Web Services (`aws.amazon.com`), where processing and memory resources are exposed through Elastic Compute Cloud instances (`https://aws.amazon.com/ec2/instance-types/`).

Monitoring and Alerting Service The Monitoring and Alerting Service provides 24/7 monitoring services on the functional and non-functional properties of the services offered by the Fredhopper Cloud Services, the service instances deployed by the Platform Service, and the healthiness of the acquired virtualized resources.

If a monitored property is violated, an alert is raised to the Cloud Engineers via emails and SMS messages, and Cloud Engineers can react accordingly. For example, if the query throughput of a service instance is below a certain threshold, they increase the amount of resources allocated to that service. For broken functional properties, such as a run-time error during service up-time, Cloud Engineers notify Software Engineers for further analysis. Fig. 4.3a shows a visualization of monitors in Grafana, the visualization framework used by ABS.

Human in the Loop

A dedicated team of Cloud Engineers is in charge of the day to day operation of the Fredhopper Cloud Services. Cloud Engineers keep track of alerts raised by the monitors and the value of monitored metrics over time. Based on their interpretation of this information, using their domain knowledge, Cloud Engineers decide if, when and how to scale up, down or restart services instances and Virtual Machines. Manual scaling rather than auto-scaling is used, as any bug or imprecision in an auto-scaling approach may have disastrous consequences:

1. Automatically scaling up too much jeopardizes the continuity of the business: the infrastructure provider charges running Virtual Machines.
2. Automatically scaling down too much may break the Service Level Agreement(s) (SLAs) between Fredhopper and customers. In the most extreme case, the web shop of a customer may become unavailable, resulting in financial and reputation damage.

The Cloud Engineers must take into account many factors when deciding if, when and how to scale. Most importantly:

- The target QoS values for service metrics specified in the SLA between Fredhopper and the customer.
- Logical and resource requirements on the deployment[3].
- General business KPIs.

[3]A deployment associates service instances to Virtual Machines

Finding scaling actions resulting in a deployment satisfying all above desiderata, and applying them at the right time is a challenging task due to several reasons.

SLAs traditionally are informal natural language documents, not represented at the software level. Thus, metrics tracked by the monitoring system (i.e., memory consumption), are not directly related to SLAs between Fredhopper and its customers. The Cloud Engineer must manually infer a relation between a combination of the metrics from the monitoring system (typically lower-level), and the metrics in the SLA (typically higher-level, aggregated at the customer level).

Synthesizing a deployment satisfying all logical and resource requirements is a computationally complex task for Cloud Engineers. Even taking only the resource requirements into consideration, it is an instance of the NP-hard multi-dimensional multi-knapsack problem, where the items are service instances (whose weights are the resource requirements for the service, like the amount of memory needed, minimal speed of CPU, etc), and the knapsacks are virtual machines. Logical requirements must also be taken into account. For example, which service instances should be co-located on the same VM, and which to deploy on a dedicated VM? For example, the Query service requires the presence of the Deployment service to function properly. Another logical requirement is to scale with multiple VMs simultaneously in different available zones (locations) in each region. This is mandated by most infrastructure providers to be eligible for compensation for faulty VMs.

In the next section we describe how HITL simulation of ABS models can be used to improve the above practice of Cloud engineers.

4.5.1 The tool

Our tool-suite for HITL simulations of Cloud services integrates several different tools.

- The SAGA tool [Boer and Gouw, 2014] was tweaked for monitoring SLA metrics and the Grafana framework visualizes the metrics
- The SmartDeployer [Gouw et al., 2016] for synthesizing deployment actions
- A logreplay tool for replaying real-world log files
- The new Haskell ABS_RT backend for real-time simulations.

We discuss below how each of these tools was exploited to contribute to the support for realistic HITL simulations.

We defined a new layered declarative generic framework in ABS which captures various monitoring concepts from QoS and SLAs to lower-level metrics, metric policies, and listenable and billable events. This framework exploits the SAGA tool for the declarative specification of service metric functions which are used to formalize SLA's. A service metric function is defined by a mapping of (time-stamped) event traces to values which indicate the different levels of the provided quality of service. These events represent client interactions with an endpoint of an exposed service API. Each monitor captures a single metric, and based on the value of that metric, suggests scaling actions to improve that metric. The MonitoringService periodically polls the registered monitors at a user-configured interval to retrieve its suggested scaling actions. An await duration(1,1) statement is used to advance the clock and determine which monitors to poll at the current time.

Our tool-suite further integrates SmartDeployer [Gouw et al., 2016] for the formalization of deployment requirements, and the automatical derivation of an executable (in ABS) provisioning script that synthesizes a deployment satisfying all specified requirements. By further integrating SmartDeployer actions into the executable, SLA-level monitors generated by SAGA, we have a formalized model that automatically suggests appropriate scaling actions at the right time: when the values of the SLA metrics give rise to it.

The simulation itself consists of replaying a log file recorded by the actual system on the ABS model of the system. The logreplay tool is responsible for firing at appropriate times a HTTP API call (as explain in section 3.2.5) to the running simulation for each request recorded in the log file. These requests will trigger ABS code that contains *Cost* annotations (Listing 4.2), which has the effect of the real-time simulation as defined for the ABS_RT backend.

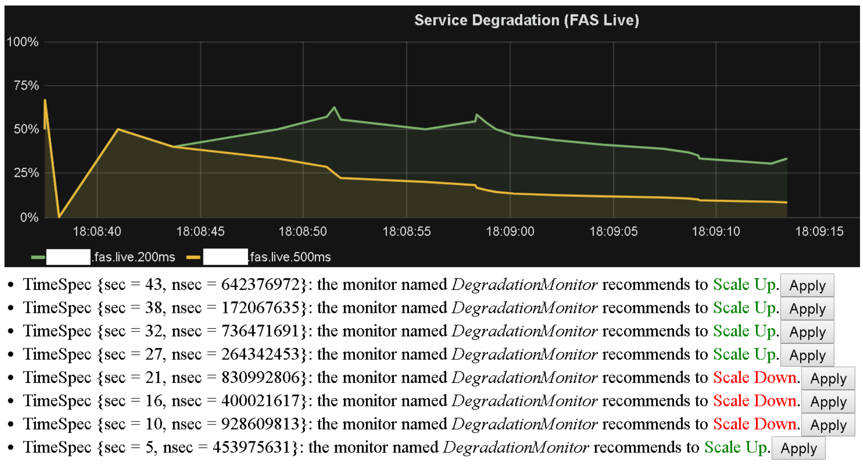
```
Bool invoke(Int request){
  print("Executing request in service:"+serviceId);
  [Cost : cost(request)] reqCount = ( reqCount + 1 );
  return True;
}
```

Listing 4.2: ABS method that process each incoming request from the log-file

This model includes automatically generated monitors in ABS which integrate the declarative specification of service metric functions of SAGA and the provisioning scripts of SmartDeployer. In the simulation, Cloud engineers can then interactively select the scaling actions recommended by the different monitors and thus acquire realtime knowledge of their consequences. In general, these selections requires specific domain knowledge which includes knowledge of past behavior. For simplicity, Cloud Engineers can interact with a running HITL simulation via an HTML/Javascript graphical user interface; a live screenshot is shown in Fig. 4.2. This interface makes also use of the HTTP API (Listing 4.3) extension as implemented in the HABS backend, for fetching the metric history and recommendations.

```
{ // ... main block header omitted
[HTTPName:"monitoringService"] IMonitoringService ms
  =new MonitoringService();
[HTTPName:"monitor1"] IDegradationMonitor dm
  =new DegradationMonitor(deployer1);
ms!addMonitor(Rule(5000,dm)); // registers a new monitor
[HTTPName:"queryService"] IMonitoringQueryEndpoint ep
  =new MonitoringQueryEndpoint(loadBalancerEndPoints,dm);
println ("Endpoints set up. Waiting for requests ...");
}
```

Listing 4.3: The main ABS block exposing the FRH services through the HTTP API.



- TimeSpec {sec = 43, nsec = 642376972}: the monitor named *DegradationMonitor* recommends to Scale Up. Apply
- TimeSpec {sec = 38, nsec = 172067635}: the monitor named *DegradationMonitor* recommends to Scale Up. Apply
- TimeSpec {sec = 32, nsec = 736471691}: the monitor named *DegradationMonitor* recommends to Scale Up. Apply
- TimeSpec {sec = 27, nsec = 264342453}: the monitor named *DegradationMonitor* recommends to Scale Up. Apply
- TimeSpec {sec = 21, nsec = 830992806}: the monitor named *DegradationMonitor* recommends to Scale Down. Apply
- TimeSpec {sec = 16, nsec = 400021617}: the monitor named *DegradationMonitor* recommends to Scale Down. Apply
- TimeSpec {sec = 10, nsec = 928609813}: the monitor named *DegradationMonitor* recommends to Scale Down. Apply
- TimeSpec {sec = 5, nsec = 453975631}: the monitor named *DegradationMonitor* recommends to Scale Up. Apply

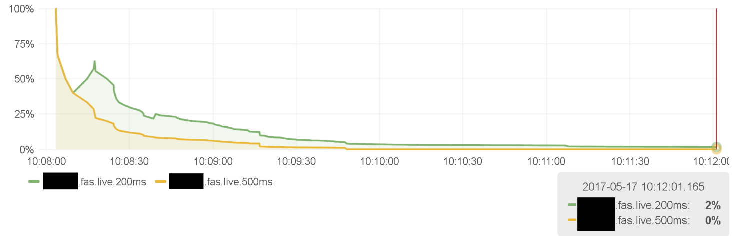
Figure 4.2: The GUI of the HITL framework intended for training Cloud Engineers.

This model-based approach of ABS and its toolset can also be used by the Cloud Engineers as a semi-automated support system: the Engineer still interacts with the Fredhopper Cloud Services to perform at the right time the desired scaling actions suggested by the framework. To achieve this the HTTP API can be used to forward queries in real-time from the production system to the ABS monitors, whereas the CloudProvider interface deploys actual IaaS virtual machines. Hence to allow the Cloud Engineer to engage in simulating real-world scenarios, or simply to interact with the system in a meaningful manner, we believe it is crucial that the simulation executes *in real-time*.

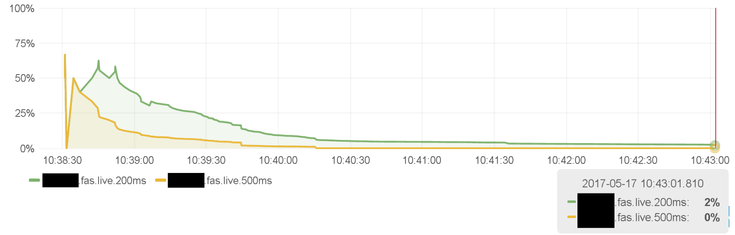
4.5.2 Benchmark

The FRH case study and its ABS model ($\approx$ 2.000 lines of code[4]) forms the basis of our experimental results. We focus on the following metric, which is part of the SLA negotiated between Fredhopper and its customers (the exact percentages are not fixed, they can be negotiated by customers):

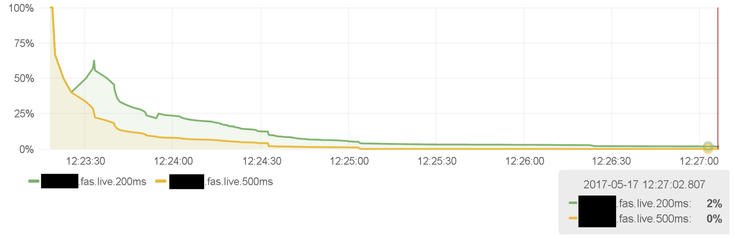
[4]The source code for the FRH model is at `http://github.com/abstools/habs-frh`



(a) Original degradation from production system



(b) Haskell simulation of the degradation when simulating the original log



(c) Erlang simulation of the degradation when simulating the original log

Figure 4.3: Degradation in the production system and as simulated on different backends

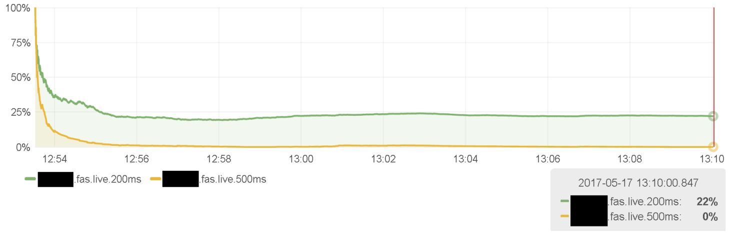
> *"Services must maintain 95% of the queries with less than 200 milliseconds of processing time, and 99% with less than 500 milliseconds, subtracting the 2% slowest queries."*

Initially, our experiments were focused on the FRH case study behavior when simulating its model (expressed in ABS) without any human intervention. A provisioning script generated by SmartDeployer automatically instantiated all services of the Cloud Architecture (Fig. 4.1), requested suitable VMs from the CloudProvider and deployed the various kinds of Service instances shown in the diagram on it. For the QueryService, a minimal setup was used with a single instance (co-located with a DeploymentService instance) deployed to an Amazon m4.large VM. The input to the simulation was a real-world log file of a particular customer with length of 4 minutes and 30 seconds, coming from a single production VM (of type m4.large). Fig. 4.3a visualizes the Service Degradation of that log file (customer names are anonymized); We then proceeded with simulating the FRH system on the Haskell and Erlang backends of ABS, inputted with the same exact log and using the same deployment scenario.

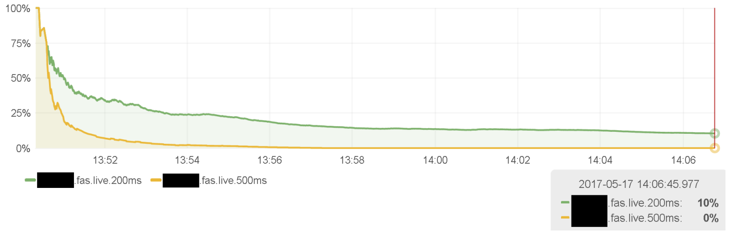
The simulation of the FRH model on the HABS backend took 4 minutes and 30 seconds to complete, which matches the log's length and encourages us to believe that the simulation is done in real-time. The output of the simulation on the HABS backend is shown in Fig. 4.3b. There is a deviation that can be seen when comparing it to the original graph of Fig. 4.3a: the HABS output reports higher degradation than what would be expected from the real-world log. This can be attributed to three causes; first, there is the overhead of processing the log file itself (network communicating to the logreplay tool). Secondly, the simulation of the real-time measurements of the log file involves *sleep* system calls, which dictates that any "sleeping" thread will be re-activated no sooner than the specified time, but most likely later than prescribed, which depends on factors such as backend implementation, hardware configuration, or the workload of the particular model. Fortunately none of these had great effect on the models we tested, and the reported degradation is negligibly affected by this. The last cause which however has a larger effect on the degradation is that the log file contains a certain number of concurrent requests (requests on a single machine that were served concurrently in time). The recorded processing time of the requests are translated into *Cost* annotations (taking into account the resource capacities of the machine that has processed the request), and therefore the concurrent execution of such requests in the simulation will further increase the *simulated* processing time of the individual requests. In general, the recorded processing time of the individual requests includes the overhead of time sharing and as such do not specify their "intrinsic" processing time. In practice we think one can obtain a "correct" model by approximating these intrinsic processing time of the individual requests by averaging over different log files and different deployment scenarios.

Moving on to the Erlang-ABS symbolic-time simulation, we observe slight inaccuracies of the output (Fig. 4.3c) compared to the original graph. These inaccuracies can be attributed to two reasons: first, the monitors act autonomously (`while (True) {await duration (1,1);...}`), so they may uncontrollably advance the symbolic time by themselves between HTTP calls of the logreplay tool; as a result the graph is slightly "stretched" because of extra erroneous time advancements. We propose two ways to mitigate this at the ABS language level: a) having a statement `every(intExp()){body};` which will register the body as a callback to be executed with the period given or b) a statement `await until(t);` which will resume the process only after the specific time given. In either case the two statements do not advance the time by themselves. The other reason which leads to inaccuracies is that the concurrent requests of the log are processed sequentially (as opposed to Haskell) because of practical difficulties of synchronizing an external tool that uses the real-world clock (logreplay) and the Erlang-ABS runtime which uses the symbolic clock. Since, as mentioned before part of the requests in the log happen to be concurrent, the resulted degradation of the Erlang-ABS simulation may differ from the expected original.

The Erlang-ABS backend took 15min and 30 seconds to complete the simulation of real-world 4min and 30 seconds of the log. This may be attributed to the fact that the granularity of the request timestamps is per *ms* (as given in the log file). We could speed it up by having a more coarse-grained (less accurate) timestamps. Furthermore, the Erlang-ABS backend does not use a (parallel) Discrete-Event simulation runtime (called also as-fast-as-possible computer simulation) but a timed-automata inspired runtime for the advancement of the clock, which requires a computationally-heavier continuous global administration of the simulation. Given the reasons above, the code for the monitors `while True {await duration (1,1); ...}` affects the execution speed. A way to mitigate this is again to have a coarser periodicity for the monitors. Based on these experimental findings, we believe in general simulation frameworks based on symbolic time are not suited for HITL simulations of Cloud applications.



(a) No scaling - 200ms metric breaks SLA



(b) Performing a Scale-up after 1 minute

Figure 4.4: No-scaling versus Scaling during the Haskell simulation

To evaluate the HITL simulation of FRH case study, a training exercise was carried out for the Cloud Engineers. Using our framework, we first visualized the Service Degradation of a different real-world log file, but include the same Service Degradation metric from the SLA as above. The deployment configuration used for that customer was the initial default configuration used by the Cloud Ops team, which provisions the minimum number of VM's, and each VM has as few resources as needed by the services running on the VM. In particular, aside from the Service instances shared between different customers, such as the PlatformService and LoadbalancerService, the non-shared initial default per-customer setup consisted of one query service instance and a corresponding deployment service instance in every availability zone (in the region of the customer), and those were deployed on an Amazon VM with instance type m4.large.

Fig. 4.4a shows the resulting Service Degradation for that customer on this deployment configuration. The graph shows that in the beginning, performance is low (and Service Degradation is high). This is caused by the fact that after a service is started, an initialization phase is triggered, and performance is (as expected) low during this phase. After a few minutes, initialization finishes and the service degradation metrics stabilize to around 20% queries slower than 200ms and 0% queries slower than 500ms (subtracting the two percent slowest queries). This means that while the target QoS as agreed in the SLA for the category "slower than 500ms" is achieved, this is (by far) not the case for the category "slower than 200ms".

After establishing that the initial default deployment configuration was not sufficient to satisfy the SLA as agreed with that customer (on that real-world query log file), the training exercise continued. The Cloud Ops were tasked with selecting and executing appropriate scaling actions to mitigate the situation. The scaling actions could be selected through the ABS HTTP API, or in a very simple front-end (Fig. 4.2).

During the training exercise, several different scenarios were trained; Fig. 4.4b shows one scenario of the effect on the Service Degradation after the engineer decided to scale up with two query services instances (and corresponding deployment service instance) in two zones on a (simulated) Amazon m4.xlarge instance after one minute (13:51) into the simulation. At time 13:54 the new machines have finished initializing, and the services deployed on them have been started. After that time, the 200ms metric quickly improves , and after about 25 minutes reaches the target $\leq$ 5% degradation.

The integrated tool suite described considerably simplified the task of the Cloud Engineers in managing the day-to-day operation of the Cloud services. In particular:

- The support for real-time simulation was critical in providing a realistic training experience for the cloud engineers. It allowed the Ops to evaluate and view metrics of the system and apply corrective actions to the system *at the same speed as they do in the production environment.*
- The high abstraction level of the metrics captured by the ABS monitoring framework enables *SLA-based scaling*, simplifying the decision process of the

Cloud ops in selecting the appropriate corrective scaling actions. Still, domain knowledge of the Cloud operator is crucial to properly "translate" their interpretation of multiple (possibly conflicting) metrics over time into corrective actions. The direct relation of the metrics to SLAs and business KPIs in our tool suite eliminated the burden on the Cloud Ops to manually interpret how traditional lower-level metrics (such as CPU usage, memory consumption) relate to the higher-level SLA/KPI metrics.

- By suggesting to the Cloud Ops only a limited number of possible corrective actions (synthesized by SmartDeployer), the number of choices the Cloud Op has to take in real-time (i.e.: which and how many services to deploy, how to link them, on what kind of VM to deploy them, etc) was reduced substantially. Since the SmartDeployer actions are synthesized based on the deployment requirements and Smartdeployer generates a corresponding provisioning script, the numerous deployment requirements are satisfied automatically "by construction". However, the quality of the suggestions (actions) proposed by the framework should be improved.

In principle, the suggested SmartDeployer scaling actions could be exploited for a full auto-scaling approach, without any human intervention. We carried out initial experiments, but it turned out to be very complex how to deal with different monitors from heterogeneous sources that give conflicting scaling suggestions, taking into account machine booting time, upcoming promotions from web-shops where peaks in demand are expected, historic data, etc. Thus keeping the human in the loop - the cloud engineers with their domain knowledge - still is crucial to optimize the day-to-day management of services.

4.6 Related Work

There exists a variety of cloud simulation tools including CloudSim [Calheiros et al., 2011], GreenCloud[Kliazovich et al., 2010], and iCanCloud [Núñez et al., 2012]; although all of these tools offer finer-grained analysis (e.g. network configuration and energy consumption in the Cloud) they rely on discrete-event computer simulation engines, which do not permit live HITL intervention on a runnning simulation. To the best of our knowledge HITL simulation of Cloud services has not been investigated before. As already stated above, HITL simulation allows Cloud engineers to acquire knowledge of the real-time consequences of their decisions directly in an interactive manner.

The Timber language [Black et al., 2002] `http://timber-lang.org` is a Haskell-like language but with strict semantics. It offers a limited form of concurrent objects with the extra feature of attaching baselines and deadlines to methods. The execution is non-deterministic and according to the Chemical Abstract Machine [Berry and Boudol, 1990].

The most known use of real-time in computing comes in the form of a real-time operating system (OS). Such OSes (e.g. QNX, FreeRTOS) use certain schedulers to maximize the responsiveness of the system, while minimizing any deadline misses: these criteria are paramount in the world of embedded systems. A real-time OS, compared to real-time language like ABS, operates on the level of system processes (executable programs) and not inside a concurrent program itself, thus it may be agnostic of any inner program characteristics.

Hardware description languages such as Verilog and VHDL and multicore simulation tools such as Graphite and Sniper are used to also construct software that “talks” about hardware. However their purposes are in contrast with the “resource-aware” programming that we detail in this chapter. Specifically, whereas such languages focus more on the description, design, architecture and implementation of computing hardware (e.g. CPUs, caches), the ABS tries to create models and simulations of software that take explicit control and monitor its hardware (cloud) resources.

On the side of ABS, related work revolves around the extension of Standard ABS with real-time concepts [Johnsen et al., 2012] and its further refinement [Bjørk et al., 2013] which adds language support for custom (user-defined) process schedulers. This refinement permits the containment of the non-deterministic scheduling of ABS processes inside the COG, programmatically; yet this is not enough to make ABS programs reproducible (deterministic), because (hard/soft)-deadline misses can still occur and more importantly there is no global — only local, inside the COG — ordering of processes.

Chapter 5

A Distributed Implementation of HABS

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

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

These cloud applications are migrated unaltered: monolithic boxes of code which are transferred from a non-cloud setting to the new cloud environment by the DevOps engineers. Such separation of the application from its execution is traditionally believed to be an advantage, long before Cloud came to existence. However, one would expect that with the introduction of the virtualized (dynamic) hardware of the Cloud, and since software logic is inherently dynamic, an application could “become aware” (as in Artificial Intelligence) and make use of its own computing power for managing its cloud resources and deployment in an optimal way, and without requiring constant administering of an engineer.

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

The proposed language is based on the *Abstract Behavioral Specification language* (ABS) that we introduced in chapter 2. We extend ABS with DC's that serve as a suitable abstraction over Cloud Virtual Machines and which allow the application to distribute itself among multiple (provider-agnostic) computing systems. The ABS developer writes code that can dynamically create, monitor and shutdown such Deployment Components (Virtual Machines) and most importantly bring up new objects inside them. To this end, an ABS cloud-application forms a cloud-aware *distributed-object* system, which consists of a number of inter-VM objects that communicate asynchronously, while recording any failures that may happen in the cloud.

An implementation of this extension must be efficient and safe so that it can be put in production code. For this, the Haskell backend of ABS (HABS) is chosen for translating ABS code to Haskell intermediate code, which is again typechecked and transformed to an executable by an external Haskell compiler. The choice of Haskell was made mainly for two reasons: the HABS backend seems to be currently the fastest in terms of speed and memory use (see Section 3.6.3, attributed perhaps to the close match of the two languages in terms of language features: Haskell is also a high-level, statically-typed, purely functional language. Secondly, compared to the distributed implementation sketched in Java, the ABS-Haskell runtime utilizes the support of Haskell's lightweight threads and first-class continuations to efficiently implement multicore-enabled cooperative scheduling. Java does not have built-in language support for ABS' algebraic datatypes; its system OS threads (heavyweight) coupled with no support for continuations make it a less ideal candidate to implement cooperative scheduling in a straightforward manner. On the distributed side, we decided against layering our solution on top of Java RMI (Remote Method Invocation) framework due to a lack of built-in support for asynchronous remote method calls and superfluous features to our needs, such as code-transfer and fully-distributed garbage collection.

We augment the HABS backend with support for *Cloud-Haskell*, a framework for type-safe, fault-tolerant distributed programming in the Haskell ecosystem. The implementation, although in its infancy, is already being tested in a real cloud environment, exhibiting promising results which are also presented. We further extend the (parallel) HABS runtime with support for Deployment Components that provide a suitable abstraction of the Virtual Machines provided by the Cloud and which allow the application to distribute itself among multiple machines. The ABS programmer can dynamically create, monitor and shutdown such Deployment Components (Virtual Machines) and most importantly assign new objects to them. As such, an ABS cloud-application consists of several inter-VM communicating objects, effectively forming a distributed-object system which can control its own deployment and still benefit from the (local) parallelism.

A Deployment Component (DC) is a minimal description of the computing resources available to an ABS application (section 4.3). We propose to extend this notion to allow any cloud resource that can properly be quantified (for example memory, disk, network, etc). On the other hand, and in contrast to the original specification, we restrict a DC to correspond solely to a Platform Virtual Machine (VM) — indeed, the terms DC and VM are used interchangeably by our extension. We call each deployed ABS application of a separate DC/VM, an *ABS node*. A running ABS program on the cloud will effectively form a distributed network of multiple inter-communicating ABS nodes.

5.1 Implementation

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

At runtime, each COG is a Haskell lightweight thread (with SMP parallelism). The COG-thread holds a process-enabled *queue*, a process-disabled *table*, and a local *mailbox*. Upon an asynchronous method call, a new process is created and put in the end of the process-enabled queue; note that processes are not threads, they are coroutines (first-class continuations) and thus can be stored as data. The COG resumes the next process from the queue until it reaches an `await` (on a future or a condition), where the process is suspended and moved to the process-disabled table. Later, another process informs the COG (by writing to its mailbox) that the await-condition is met; the COG will move back the process to the enabled queue. This strategy avoids *busy-wait* polling the boolean await conditions of processes. For more information, we refer to the section 3.5 where the runtime execution of HABS is detailed.

Moving on to distributed part, the distributed communication of ABS processes is realized by Cloud-Haskell [Epstein et al., 2011], which is a Haskell library for type-safe, fault-tolerant distributed programming. The distribution model of Cloud-Haskell resembles that of the Erlang programming language, with the difference being that Cloud-Haskell has extra support for type-safe and version-safe message passing, features that we also make very much use of in our Cloud Runtime extension. We reuse the network transports and serialization protocols defined in Cloud Haskell for the ABS transmitted data between Virtual Machines. Each COG-thread is accompanied by a separate Cloud-Haskell thread (also lightweight) that listens for messages in a *public* mailbox and *forwards* them to the local mailbox of its associate COG-thread. This approach was chosen to firstly, avoid needless network-serialization between local communication and secondly, treat our distributed extension as optional to our (previously SMP-only) Haskell backend. The resulting remote communication remains transparent to the user: new objects can be remotely created inside a different machine and asynchronous calls are made to remote objects (living inside a remote machine) without changing the syntax and semantics of the ABS language.

As we have seen previously in chapter 4, a minimal implementation of DC has functionality for (1) shutting down the corresponding virtual machine and (2) probing for its average *system load*, i.e. a metric for how busy the system stays in a period of time. We use the Unix-style convention of returning 3 average values of 1, 5 and 15 minutes. In the case of (1), a VM shutdown implies that its cloud resources are eventually freed. Each kind of cloud infrastructure service carries its own implementation of DC. The intention is that the user will not have to provide such class implementations since the implementation of deployment components will come bundled with class libraries for common cloud-backend technologies (Amazon, OpenStack, Azure, etc).

The newly-created object will "live" in the specified DC. The annotated [DC: ..] new behaves similar to the new keyword: it creates a new COG, initializes the object, and optionally calls its run method. The annotation [DC:] new Class returns a *remote object reference*, (also called a proxy object; o1 in the above example) whose methods can be called asynchronously and which can passed around in parameters as normal. Every remote object reference is a single "address" *uniquely identified* across the whole network of nodes. Calls to [DC: ...] new will also (besides shutdown, load) block until the VM is up and running. From a theoretical standpoint, a remotely-spawned object must point to the same code (attributes and methods) as in a local object.

5.1.1 Connection to Cloud infrastructure

To properly support resource-aware programming (letting an ABS model manage and control its resources), when creating a new DC, a cloud provider is contacted in the background (usually via an XML-RPC API) and asked to bring up a new VM with the given characteristics. After the machine has booted, the caller replicates itself (the current ABS application) by transmitting its machine code to the newly-created machine. In case the cloud provider offers heterogeneous platforms (different OS or CPU architecture), we instead transmit the ABS source code and compile it in-place with our compiler toolset (that resides beforehand in the VM's image). The new machine runs the transmitted ABS application and sends an acknowledgment signal to its creator, which can now start computations to the new DC by spawning new objects in it. The image of the virtual machine that is chosen is pre-defined (via a specific tag name in the cloud infrastructure) and should come with the appropriate Haskell and HABS compiler toolset already installed. Uploading the image is the only manual step required by the cloud ABS user.

The creation of Deployment Components is done under the hood by contacting the corresponding (cloud) platform provider to allocate a new machine, usually done through a REST API. The executable is compiled once and placed on each created machine which is automatically started as the first user process after kernel initialization of the VM has completed.

Simply put, a DC corresponds to a single Virtual Machine having certain resources which executes ABS code. We restrict the definition of DC to correspond only to a Platform Virtual Machine (VM)[1] residing inside the boundaries of a Cloud infrastructure.

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

```
module StandardLibrary.SomeProvider;

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

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

The implementer can expose other properties to DCs, such as, network, number of IO operations, VM region location. A concrete implementation, which is omitted for brevity, usually involves some high-level ABS logic coupled with low-level code written in a foreign language (in our case Haskell). The average ABS user will not have to provide such connections to the cloud, since we (the implementers) intend to provide class implementations for most major cloud providers/technologies, in an accompanying ABS library. With this approach, we lift the low-level API of the cloud provider (usually XML-RPC) to a *typed* high-level API (e.g. CpuSpec and MemSpec datatypes).

[1]A platform virtual machine "emulates the whole physical computer machine, often providing multiple virtual machines on one physical platform" (from Wikipedia

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

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

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

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

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

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

Currently we are investigating the migration of ABS processes between DCs (machines); this can theoretically be achieved since ABS processes are merely data, and thus can be serialized and remotely transferred (migrated) from machine to machine.

5.1.2 Serialization

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

Cloud Haskell code is employed for remote method activation and future resolution: the library provides us means to serialize a remote method call to its arguments plus a static (known at compile time) pointer to the method code. No actual code is ever transferred; the active objects are serialized to unique identifiers across the entire network and futures to unique identifiers to the caller object (simply a counter). The serialized data, together with their types, are then transferred through a network transport layer (TCP, CCI, ZeroMQ); we opted for TCP/IP, since it is well-established and easier to debug. The data are de-serialized on the other end: a de-serialized method call corresponds to a continuation which will be pushed to end of the process queue of the callee object, whereas a de-serialized future value will wake up all processes of the object awaiting on that particular future.

5.1.3 Garbage Collection

In a local-only setting, all ABS-based values, i.e. ADTs, futures, objects are automatically garbage-collected by the underlying Haskell GC. However, in our distributed setting some object/future references may have to be transmitted outside as proxy references, which results to the local ABS system garbage-collecting "too-early". An obvious solution would be to abolish automatic GC altogether, but that would hinder the development of software applications, especially those supposed to be long-running (as is the norm in cloud applications). On the other hand, introducing *distributed garbage collection* to ABS would allow both local and remote objects to be automatically GC'ed. The downside is that it is much more complex for the user to reason about the GC-incurred performance penalty which may be considerable. We chose a middle ground, where objects are by default GC-enabled and only become disabled when they are remotely communicated over (to another DC). The implementation has been straight-forward: a process appends the local object reference(s) that are transmitted remotely to a locally-held list of GC-disabled objects. This global list is held during the lifetime of the node, effectively surpassing the Haskell's garbage collector underneath. Our design choice was based on best practice; we believe that a distributed cloud ABS application of many DCs would contain a combination of a lot of local ephemeral objects, and only few long-lived remote objects.

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

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

5.1.4 Failures in the Cloud

In cloud computing, and in any distributed system in general, failures are more frequent, mostly because of unreliable networks. Based on this fact, we further extend ABS with proper support for extensible, asynchronous exceptions. At the language level, exceptions are pure expressions modelled as single-constructor values of the ADT `Exception` , as detailed in section 3.2.1. To define new exceptions the user writes a declaration similar to an ADT declaration, e.g. `exception MyException(Int, List<String>);`. Our cloud extension predefines certain common "local" exceptions (e.g. NullPointerException, DivisionByZeroException) and cloud-related exceptions (e.g. NetworkErrorException, DCAllocationException, DecodingException).

Normally, if an exception reaches the outermost caller without being handled, its process will stop. We introduce a special built-in keyword named die that changes this behaviour and causes an object to be nullified and *all* of its processes to stop. With this in hand, a distributed application can easily model objects that can be remotely killed:

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

Note that like Cloud-Haskell and unlike (distributed) Erlang, if a network error occurs between computing nodes, the connection will be dropped and not automatically re-connected. Unlike Cloud-Haskell, there is currently no primitive operation in HABS to allow the re-connection to a node after a network failure.

5.2 Extension: Service Discovery

Service discovery, the dynamic acquisition of a computing resource suitable to fulfill a specific task or group of tasks (i.e. a service), can help to decouple parts of a large distributed system. As such, service discovery is of interest to the Envisage case studies since certain large, distributed system architectures can be modelled naturally in this way. This section first briefly explains the basics of service discovery and lays out the design criteria for integrating service discovery into the ABS language.

In its most basic form, we see a service as a computing entity suitable to fulfill one or more specific tasks. Since in ABS tasks are modelled via method calls, it makes sense to model services as ABS interfaces and implement them using ABS objects. Note that in conventional object-oriented languages, objects and interfaces might not be sufficient, but the ABS concepts of asynchronous calls, distribution via deployment components, and safe parallel execution make ABS objects powerful enough to become services.

We augment the feature of "Deployment Components" (DC) with the ability of discovering available services offered by a DC. We adopt the notion of a service being represented by an ABS interface.

The acquire , expose, and unexpose methods are added to the DC interface. Thus, the DC interface becomes:

```
interface IDC {
    Unit shutdown();
    Triple<Rat,Rat,Rat> load();
    A acquire<A>(A);
    Unit expose<A>(A);
    Unit unexpose<A>(A);
}
```

The newly-introduced methods are parametrically-polymorphic; the programmer will instantiate their types when using them, as the following example:

```
{
DC dc1 = new NebulaDC(...);
MailService mail_server ;
Fut<MailService> f = dc ! acquire( mail_server );
mail_server = f.get ();
mail_server ! send_mail (...);
}
```

The acquire method takes as input a "phantom object". The object is called phantom since the object's contents or reference are not actually send; the object is there only to give hints to the ABS compiler of what is the Interface we want the acquired object to comply with. The phantom object can also be introduced with a (nullary) declaration, as in the second line above: MailService mail_server ;

The call to acquire makes a request to the DC, asking for a reference to an (possibly remote) object that complies to the IDC interface/service. Upon processing the request, the DC searches through its *directory facility* for object subscriptions that support such an interface. If there is no search match, the DC will raise the ServiceNotFoundException and record it in the future as a fault. If the match succeeds, a reference to a complying object is returned. The returned object reference from the call to acquire can then be assigned back to the phantom object or any other object. This returned object reference is typed exclusively by the mentioned Interface and the user cannot normally know which is the actual class name (class implementation) behind it, unless this can be guessed through a method implementation.

An object can be subscribe to any DC's *directory facility* through the expose method. For example:

```
WebService ws = this;
dc ! expose(ws);
```

Accordingly, an object can be unsubscribed for some of its services/interfaces with the unexpose method:

```
AdminService a = admin_object;
MyInterface m = this;
dc ! unexpose(m);
dc ! unexpose(admin_object);
```

Following the approach of phantom objects, the arguments to expose and unexpose are type-checked with respect to the available interfaces that the object (class) implements. If the programmer omits such a phantom definition, the compiler will compute the object's principal interface (the object passed to acquire, expose, unexpose) through type-checking. If the ABS compiler cannot compute such a principal interface, it emits a type-checking error.

This peculiar design choice (of phantom objects) was made so as to not introduce any backwards-incompatibilities (adding interfaces as first-class citizens) and further more built-in keyword-statements. A further advantage is that the implementations of acquire and (un)expose methods can vary between DCs and thus be specific to the underlying service discovery technology of the cloud provider.

Two-times (un)exposing will not yield a runtime exception and will be silently suppressed. Each DC keeps track of its own subscribed objects and automatically unsubscribes them in case they fall out of context, i.e. they have normally or exceptionally terminated.

5.3 Experiments and Results

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

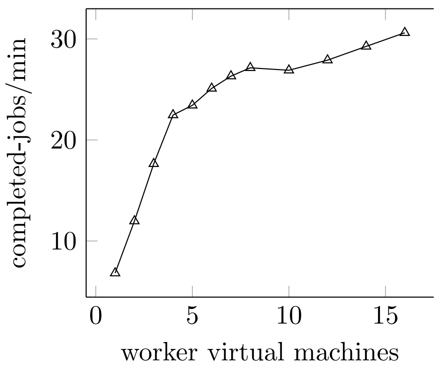
The *static* load-balancer case is a fairly straight-forward cloud ABS application, consisting of 3 classes of LoadBalancer, Worker, and Client, exchanging asynchronous method calls of job requests/results. The LoadBalancer runs the main block and initially creates N number of Worker DCs (VMs) before starting to accept requests and forwarding them to workers in round-robin. We ran this static deployment against varying size (N=1..16) of worker VMs. The results of the runs are shown in Figure 5.1(a) stripped from the initial boot time of VMs. What we can draw from these results is that the completed jobs (per minute) nearly doubles when we double the number of worker VMs until we reach 5 workers. After that, we still increase the completed jobs but with a slower pace. This observation can be attributed to the fact that a point is reached where there is not a significant benefit from adding more worker VMs; the rate of job requests is always steady, thus worker VMs are "slacking".

We modified the static load-balancer to an *adaptive* version, that takes full advantage of the expressiveness of the cloud extension. The LoadBalancer creates now only one VM initially. We accommodate the LoadBalancer with a HeartBeater object which periodically retrieves the load from each worker in the VM "farm". The HeartBeater computes the average `load` of all VMs and if this average exceeds 80%, it creates a new DC (VM), adds it to the current farm, and remotely spawns a Worker in the new DC. We illustrate a particular run of this configuration in Figure 5.1(b) (NB: VM boot times are not subtracted from the result). Each asterisk $*$ in (b) is a point where the HeartBeater decides to create a new DC. This run stabilizes on 6 workers, which is a good approximation of maximum performance (according to Figure 5.1(a)), and possibly a good choice if we took into account any VM costs. As an extra, the HeartBeater could potentially `shutdown` machines if their load remained small (under a threshold) for a certain time.

The tests were conducted on the SURF cloud-provider with OpenNebula IaaS, on VMs with modern 8-cores, each with 8GB RAM and 20Gbps Ethernet. Interesting to mention is that each worker can benefit from ABS multicore (SMP) parallelism. A snippet of the HeartBeater follows with the full ABS code at our repository[2]:

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

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



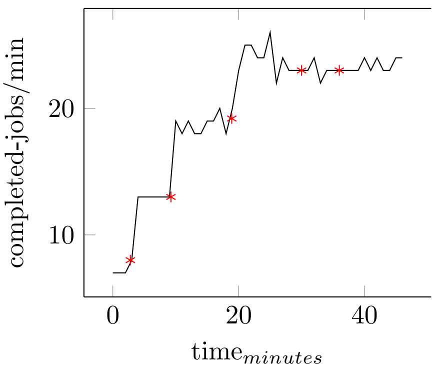


Figure 5.1: (a) Static deployment of VMs (b) Adaptive Deployment over time

For future work we are considering additions both at the language and runtime level. At the language level, it would be beneficial to include, besides the system load, other metrics such as memory, disk usage, object count, process count, exceptions raised (as partly done in the cloud simulation of Section 4.5). In this way, an ABS application would enhance its monitor and cloud-control logic. In a different direction, we plan to work on adding a basic *service discovery* mechanism to the standard library of ABS, as proposed by section 5.2, i.e. by extending the DC interface with two extra methods: an `acquire(Interface obj)` method that returns a reference to a remote object implementing the provided `Interface`; an `expose(Interface obj)` that subscribes the passed object together with its current interface-view to the service registry of the DC.

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

5.4 Case Study: Distributed Preferential Attachment

We ran the ABS-Haskell implementation of the PA algorithm by varying the graph size, on a distributed cloud environment kindly provided by the SURF foundation. The hardware consisted of identical virtual machines interconnected over a 10Gbps ethernet network; each Virtual Machine (VM) was a a single-core Intel Xeon E5-2698, 16GB RAM running Ubuntu 14.04 Server edition. The runtime execution results are shown in Fig.5.2; the execution time decreases while we add more VMs to the distributed system, which suggests that the distributed algorithm scales. However, even with 8 Virtual Machines the implementation cannot "beat" the execution time of one VM running PA sequentially; to achieve better performance, we may need to include more VMs. The reason for this can be attributed to the significant communication overhead, since each worker will send a network packet for every request call made.

On the other hand, the memory consumption (Table 5.1) is more promising: a larger distributed system requires less memory per VM. For example with the largest tested graph size, a distributed system of 8 VMs requires approx. 2.5 times less memory per VM than a local system. This allows the generation of much larger PA graphs than would otherwise fit in a single machine, since the graph utilizes and is "distributed" over multiple memory locations.

To improve the execution performance and time scaling, we further refined our approach to solving the PA problem by combining multiple request messages in a single TCP segment; this change increases the overall execution performance by having a smaller overhead of the TCP headers and thus less network communication between VMs, and better network bandwidth. In another (orthogonal) direction, we could utilize the many cores of each VM to have a parallel-distributed hybrid implementation in ABS-Haskell for faster PA graph generation, but this is left for future work.

This new improved version of the distributed PA algorithm is implemented as well in distributed HABS, [Bezirgiannis and Boer, 2016], with small high-level changes

	Total number of VMs			
Graph size	1	2	4	8
$n = 10^6, d = 3$	306	423	313	229
$n = 10^6, d = 10$	899	1058	644	411
$n = 10^7, d = 3$	1943	2859	1566	874
$n = 10^7, d = 10$	6380	9398	4939	2561

Table 5.1: Maximum memory residency (in MB) per VM.

	Total number of VMs				
Graph size	1	2	4	8	16
$n = 10^6, d = 3$	306	266	212	155	114
$n = 10^6, d = 10$	899	1028	547	354	221
$n = 10^7, d = 3$	1943	3242	1603	967	621
$n = 10^7, d = 10$	6380	9668	6702	3611	1905

Table 5.2: Maximum memory residency (in MB) per VM (refined approach).

to the model. Beside higher level of abstraction at the programming level thanks to our proposed improvement, the distributed runtime system provides more than $6x$ speedup performance compared to the same implementation without using the improvement, presented in [Azadbakht et al., 2017a]. The results of the refined approach are illustrated in Fig. 5.3. The distribution overhead increases the execution time for two machines, which is compensated by the parallelism achieved through adding more VMs. We managed this time to achieve positive speedup compared to a sequential algorithm implementation of one local machine, when using greater or equal to 8 virtual machines for distributing workload. As shown in the new memory results of Table 5.2, it is still the case that the memory consumption decreases by adding more VMs, which enables generating extra-large graphs which cannot fit in centralized-memory architectures.

5.5 Related Work

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

5.5.1 Distributed programming languages

Erlang is one of the first distributed-oriented languages that next to the canonical message-passing communication, offers distinct features, such as hot-code loading and binary serialization of arbitrary closures — thus the capability to transfer them *over the wire*. This comes with a cost in safety since the serialized Erlang data are untyped and usually unversioned. The Akka framework brings distributed actors to the Scala language. Although Akka provides a rich library and toolkit, it currently lacks a cloud-aware API. At runtime The Java RMI (Remote Method Invocation) is a library bundled in the Java platform for communication between remote objects. The product pioneered in areas such as bytecode downloading and distributed-GC. The method invocation is strictly synchronous (the caller has to wait for the remote method to finish) and thread-unsafe. JADE[Bellifemine et al., 1999] is an active distributed multi-agent system also built in Java; agents are more expressive than actors at the expense of program complexity and, possibly, performance.

5.5.2 Cloud middleware and management

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

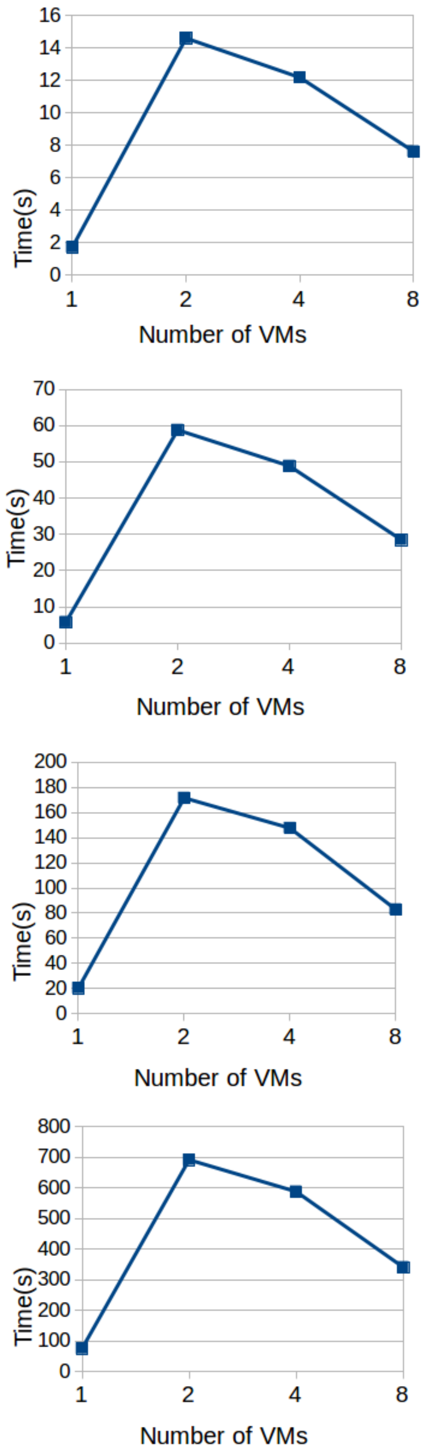
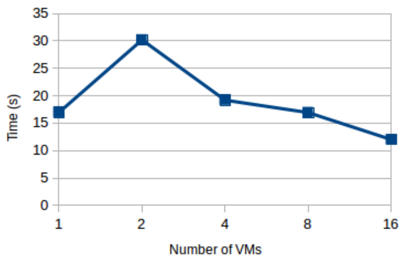


Figure 5.2: Performance results of the distributed PA in ABS-Haskell for graphs of $n = 10^6$ nodes with degree $d =$ (a) 3 , (b) 10 and $n = 10^7$ nodes with degree $d =$ (c) 3 , (d) 10.



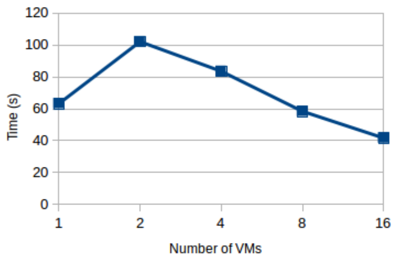


Figure 5.3: Performance results of the refined distributed PA in ABS-Haskell for graphs of $n = 10^7$ nodes with $m =$ (a) 3 , (b) 10.

Chapter 6

Conclusion and Future Work

We have presented the modeling language ABS and its implementation, that tries to engage in all the three problems of software-engineering, namely performance, security and complexity. We more or less focused on execution performance, because we believe that this is under-represented in the research field of modeling languages, and formal methods in general. However, through our experiments and case studies we did investigate on matters of complexity (e.g. the preferential attachment problem) by manually proving the correctness of a subset of our implementation.

More specifically, in Chapter 3 we have presented a concurrent, object-oriented language (ABS) and its compilation to Haskell using continuations. The runtime utilized Haskell-GHC' lightweight (green) threads to automatically benefit from any multicore parallelism. The compilation to a subset of ABS is formalised in order to establish that the program behaviour and the resource consumption are preserved by the translation. Compared to the only other formalised ABS backend [Johnsen et al., 2010a] (in Maude), our Haskell translation admits the preservation of resource consumption, and as a side benefit, makes uses of an overall faster backend.[1] The performance that the backend promises has been shown through a list of micro-benchmarks and real-world cases of the cache coherence protocol and preferential-attachment implementations.

In Chapter 4, we integrate the timed extension of ABS into HABS with its real-time characterization of the abstract time. The virtualized resources and systems (deployment components) are also present in HABS as first-class citizens of the language. We made use of these new modeling constructs to build an industrial case

[1] `http://abstools.github.io/abs-bench` keeps an up-to-date benchmark of all ABS backends.

study for the human-in-the-loop simulation of virtualized (cloud) services. Our initial experimental results on the use of the presented tool-suite provides clear evidence for the viability of human-in-the-loop (HITL) simulation of Cloud services for training purposes. The training sessions themselves can further be used to provide feedback to the underlying ABS models of the Cloud services and the monitors. Ultimately, the resulting fine-tuning of these models may reach a level of maturity and confidence that allows their deployment in the real-time monitoring and management of the actual service instances. In general, we believe that HITL simulation of Cloud services provides a variety of interesting and challenging research problems, for example mining the log files to calculate an approximation of the "intrinsic" processing time of the individual service requests, cancelling the effect of time sharing.

In Chapter 5, we presented an extension to the ABS language that permits the management of an application's own cloud-deployment inside the language itself. We discussed the realization of such extension (by a Haskell transcompiler) and the execution of an ABS cloud application (based on Cloud-Haskell). Results showed that ABS can benefit from the extra performance that the Cloud offers. Moreover, the extension gives to ABS the expression power it needs to fuse the application logic with the application's own (dynamic) deployment logic. A positive side-effect of the proposed extension is that, ABS being primarily a modeling language, could now be used to model also an application's deployment. Indeed, such cloud-aware software models could be simulated against different and dynamically-varying cloud deployment scenarios. We believe that the cloud extension of ABS leads to new opportunities for furthering the application of formal methods to cloud computing, for example: specifying, verifying, and monitoring Service Level Agreements (SLA) of software systems — with that being the overall goal of ENVISAGE, our current research project. Indeed, we like to envisage software that is aware of its deployment and thus can control it, while its users merely monitor its behaviour via SLAs signed between the interested parties.

6.1 Future Work

There are multiple directions to take for improving in the future the HABS language and runtime system: in the front of the parallel runtime library, the resource-modeling and simulation, and/or the distributed part of the HABS framework.

The parallel version of HABS presented in Chapter 3 has been extensively studied and tested through benchmarks, experiments and case studies. Also, a simplification of the runtime library has been formally shown to preserve the correctness and resources during the translation of a subset of ABS to the Haskell functional language. However, in real execution we rely on the lower-level C-written GHC runtime system, which we have little guarantees about. A solution to this would be to utilize an approach of systematic testing, as done in [Walker and Runciman, 2015], to test if our HABS runtime library maintains certain progress properties and guarantees by exhaustively testing the threading non-determinism.

Regarding the resource-aware modeling of ABS through the Timed-ABS extension and the simulation of cloud resources, an interesting area to investigate is the support of ABS for discrete-event simulation. We envision a design of such added support where methods correspond to discrete events of the simulation and method calls are merely the "firing" of such events.In this regard, since the "firing" of events happens at specific (as in discrete) time, each ABS method call has to be *annotated* with a timestamp, for example `[Time: t1] o!m1();`. The method's parameters become arguments to such events and as such can be seen as events (instances of events) themselves. Assuming that the events can be ordered (lexicographic ordering of method names) as well as their arguments (data-specific ordering) and since we can provide a fixed ordering (across any successive executions of the same program) of COG identities, we can guarantee the reproducibility of such discrete-event simulations done with the timestamp-augmented extension of the ABS concurrent language. For the implementation side (i.e. the simulation engine), there has been extensive literature for the advancement of algorithms for executing such simulations; two main categories of fast simulation-engine algorithms, which can also benefit from parallel and distributed execution, have been established: that of conservative class algorithms found in [Misra, 1986], and that of optimistic algorithms first proposed in [Jefferson and Sowizral, 1985]. Still, however, there is no clear winner between those categories which besides the common trade-off between execution time& memory, is very much dependent on the specific model that is simulated.

The distributed runtime of HABS detailed in Chapter 5 may receive for the future multiple optimizations that can make the distributed computing in ABS both faster and more robust. As a first optimization, we consider the propagation of futures to their holders. Currently, any resolved future has to be asked for its value directly to its resolver (the method's callee). Since future values are immutable, a first optimization would be instead of contacting the resolver itself is to contact the specific Deployment Component that passed the future onto the holder in a propagation-like fashion. On a similar front, if the deployment component of the resolver of the future fails (because of an exception or hardware error), the value of the future will not be available to its holder anymore, even if its (immutable) value has be computed (resolved). To solve this, we could introduce a way of many-copy replication and caching of the future values among the members (Deployment Components) of the distributed system. In this way, the holder of futures can fetch their values even after the error of their original Deployment Components.

Finally, the distributed version of HABS would greatly benefit from experimenting with a large computing scenario, involving perhaps a larger scale version of the Preferential Attachment case study (with many computing nodes) or some analysis of big-data; the ABS language is a good fit for expressing the concurrent aspects of these analyses and, moreover, the management of the cloud infrastructure can become easier with the cloud extension of ABS. A testing of such a large scale can give us more confidence of the robustness and readiness of the distributed HABS platform.

Acknowledgements

I would like to thank my fiancée Joëlle who was always there for me. I also like to thank my mother Kaiti, father Antonis, and sister Rina for supporting me from abroad. Thanks goes to my “clean parents” (schoonouders) and Jeske for de gezelligheid.

Finally, I want to thank some friends for being there for me before and during my PhD: Charalampos Kiskinis, Christos (Ntempa) Raptis, Christos Orlis, Stergios Gkatsis, Thomas Taskoudis.

Summary

The physical constraints in computer hardware manufacturing and the industry's eagerness to keep up with the Moore's law, led to the establishment of multicore processors and (Cloud) distributed systems in our everyday use. To be fully utilized, these technologies of "simultaneous" processing often require elaborate modifications to the computer software. This has placed a large burden on the software-development side, especially when also taking into account the ever-increasing demand for more featureful, thus likely more complex software. One particular method to tackle software complexity is software modeling, which leaves out certain implementation details and focuses instead on the functional correctness of the software. Yet, to gain full performance from the aforementioned advancements in hardware, a model of software should be aware of the hardware resources, at least in an abstract manner.

In this thesis, we strive to address this challenge by constructing a modeling language to write software which can take advantage of recent hardware developments (multicore, cloud) without compromising in its abstraction levels. Our language is based on top of the Abstract Behavioral Specification (ABS), which is an executable modeling language with a focus on cooperative-multitasking concurrency. We translate programs written in our ABS-based language to Haskell, an established functional programming language, since Haskell comes with built-in support for both cooperative concurrency in the form of coroutines, and multicore parallelism through lightweight threads of execution. Further, we formally prove the correctness as well as the resource-consumption preservation of the translation of a subset of our language to Haskell. To put our solution to test, we compare its performance to other existing ABS-based implementations.

To enable software models take control of their computing resources, we extend our language with certain constructs that abstract (virtualize) over the hardware. This "resource-aware" language extension is packaged in a tool-suite for human-in-the-loop simulation of Cloud services; such a live simulation can be used for training DevOps engineers to the cloud environment of IT companies.

Finally, we provide an implementation of distributed communication and a connection to the Cloud infrastructure, so that software models written in our language can be executed as distributed applications. Because models are "resource-aware",

they can programmatically monitor and control their own Cloud deployment. Our implementation is the first realization of the earlier “Deployment Components” concept of ABS to abstract over Virtual Machines of the Cloud and enable any ABS application to distribute itself among multiple Cloud-machines.

Samenvatting

De fysieke beperkingen in de productie van computerhardware en het verlangen van de industrie om de wet van Moore bij te houden, hebben gezorgd voor het gebruik van multicore processoren en gedistribueerde systemen (Cloud) in ons dagelijks leven. Deze technologien van “simultane” verwerking vereisen vaak ingrijpende aanpassingen in de computersoftware om volledig te worden benut. Dit legt een zware last op softwareontwikkeling, zeker als er rekening gehouden wordt met de voortdurend toenemende vraag naar functionelere en waarschijnlijk complexere software. Een van de methodes om softwarecomplexiteit aan te pakken, is door modellering van software. Deze methode laat irrelevante implementatiedetails weg en legt daarbij de nadruk op de functionele juistheid van de software. Echter, om de eerdergenoemde verbeteringen in hardware optimaal te benutten, moet een softwaremodel bewust zijn van de systeembronnen, in ieder geval op een abstracte manier.

In dit proefschrift proberen we deze uitdaging aan te pakken door een eigen modelleertaal te ontwerpen. Deze taal genereert software die kan profiteren van recente hardware ontwikkelingen (multicore, cloud), zonder afbreuk te doen aan zijn abstractieniveau’s. Onze taal is gebaseerd op de Abstract Behavioral Specification (ABS). Concurrency in deze modelleertaal focust op de coperatieve multitasking. Programma’s geschreven in onze ABS-gebaseerde taal worden vertaald naar de functionele programmeertaal Haskell, vanwege de ingebouwde ondersteuning in Haskell voor zowel de coperatieve multitasking in de vorm van coroutines, als het multicore parralelisme door lichtgewicht uitvoeringsthreads. Daarnaast bewijzen we formeel zowel de juistheid van de vertaling naar Haskell als het behoud van de resource consumptie na vertaling in een deel van onze taal. Om onze oplossing te testen, vergelijken we zijn prestatie met andere bestaande ABS-gebaseerde implementaties.

Om softwaremodellen in staat te stellen om de controle over hun computingbronnen te nemen, breiden we onze taal uit met verschillende constructies die een abstractie maken over de hardware. Deze uitbreiding van de “bronbewuste” taal is onderdeel van een nieuwe methode voor human-in-the-loop simulaties van Clouddiensten. Zo’n simulatie kan gebruikt worden voor training van DevOps-ingenieurs in de cloudomgeving van IT-bedrijven.

Tot slot bieden we een implementatie aan voor gedistribueerde communicatie en een verbinding met de Cloud infrastructuur, zodat softwaremodellen die geschreven zijn in onze taal uitgevoerd kunnen worden als gedistribueerde applicaties. Omdat modellen “bronbewust” zijn, kunnen ze programmatisch hun eigen cloud resource infrastructuur monitoren en beheren. Onze implementatie is de eerste realisatie van het eerdere concept van “Deployment Componenten” van ABS die een abstractie te representeren van Virtuele Machines in de Cloud en om elke ABS-applicatie in staat te stellen zich te distribueren onder meerdere Cloud-machines.

Bibliography

[Albert et al., 2015a] Albert, E., Arenas, P., Correas, J., Genaim, S., Gómez-Zamalloa, M., Martin-Martin, E., Puebla, G., and Román-Díez, G. (2015a). Resource Analysis: From Sequential to Concurrent and Distributed Programs. In *FM 2015: Formal Methods*, volume 9109, pages 3–17. Springer International Publishing, Cham.

[Albert et al., 2015b] Albert, E., Arenas, P., Correas, J., Genaim, S., Gómez-Zamalloa, M., Puebla, G., and Román-Díez, G. (2015b). Object-sensitive cost analysis for concurrent objects. *Software Testing, Verification and Reliability*, 25(3):218–271.

[Albert et al., 2014a] Albert, E., Arenas, P., Flores-Montoya, A., Genaim, S., Gómez-Zamalloa, M., Martin-Martin, E., Puebla, G., and Román-Díez, G. (2014a). SACO: Static Analyzer for Concurrent Objects. In *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 562–567. Springer, Berlin, Heidelberg.

[Albert et al., 2012] Albert, E., Arenas, P., and Gómez-Zamalloa, M. (2012). Symbolic Execution of Concurrent Objects in CLP. In *Practical Aspects of Declarative Languages*, volume 7149, pages 123–137. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Albert et al., 2015c] Albert, E., Arenas, P., and Gómez-Zamalloa, M. (2015c). Test Case Generation of Actor Systems. In *Automated Technology for Verification and Analysis*, Lecture Notes in Computer Science, pages 259–275. Springer, Cham.

[Albert et al., 2016] Albert, E., Bezirgiannis, N., Boer, F. d., and Martin-Martin, E. (2016). A Formal, Resource Consumption-Preserving Translation of Actors to Haskell. In *Logic-Based Program Synthesis and Transformation*, Lecture Notes in Computer Science, pages 21–37. Springer, Cham.

[Albert et al., 2014b] Albert, E., Boer, F. S. d., Hähnle, R., Johnsen, E. B., Schlatte, R., Tarifa, S. L. T., and Wong, P. Y. H. (2014b). Formal modeling and analysis of resource management for cloud architectures: an industrial case study using Real-Time ABS. *Service Oriented Computing and Applications*, 8(4):323–339.

[Albert et al., 2013] Albert, E., Flores-Montoya, A., Genaim, S., and Martin-Martin, E. (2013). Termination and Cost Analysis of Loops with Concurrent Interleavings. In *Automated Technology for Verification and Analysis*, volume 8172, pages 349–364. Springer International Publishing, Cham.

[Azadbakht et al., 2017a] Azadbakht, K., Bezirgiannis, N., and Boer, F. S. d. (2017a). Distributed Network Generation Based on Preferential Attachment in ABS. In *SOFSEM 2017: Theory and Practice of Computer Science*, Lecture Notes in Computer Science, pages 103–115. Springer, Cham.

[Azadbakht et al., 2017b] Azadbakht, K., Bezirgiannis, N., and Boer, F. S. d. (2017b). On Futures for Streaming Data in ABS. In *Formal Techniques for Distributed Objects, Components, and Systems*, Lecture Notes in Computer Science, pages 67–73. Springer, Cham.

[Azadbakht et al., 2016] Azadbakht, K., Bezirgiannis, N., Boer, F. S. d., and Aliakbary, S. (2016). A High-level and Scalable Approach for Generating Scale-free Graphs Using Active Objects. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 1244–1250, New York, NY, USA. ACM.

[Barabási and Albert, 1999] Barabási, A.-L. and Albert, R. (1999). Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512.

[Bellifemine et al., 1999] Bellifemine, F., Poggi, A., and Rimassa, G. (1999). JADE–A FIPA-compliant agent framework. In *Proceedings of PAAM*, volume 99, page 33. London.

[Berry and Boudol, 1990] Berry, G. and Boudol, G. (1990). The Chemical Abstract Machine. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 81–94, New York, NY, USA. ACM.

[Bezirgiannis and Boer, 2016] Bezirgiannis, N. and Boer, F. d. (2016). ABS: A High-Level Modeling Language for Cloud-Aware Programming. In *SOFSEM 2016: Theory and Practice of Computer Science*, Lecture Notes in Computer Science, pages 433–444. Springer, Berlin, Heidelberg.

[Bezirgiannis et al., 2017] Bezirgiannis, N., Boer, F. d., and Gouw, S. d. (2017). Human-in-the-Loop Simulation of Cloud Services. In *Service-Oriented and Cloud Computing*, Lecture Notes in Computer Science, pages 143–158. Springer, Cham.

[Bijo et al., 2016] Bijo, S., Johnsen, E. B., Pun, K. I., and Tarifa, S. L. T. (2016). A Maude Framework for Cache Coherent Multicore Architectures. In *Rewriting Logic and Its Applications*, Lecture Notes in Computer Science, pages 47–63. Springer, Cham.

[Bjørk et al., 2013] Bjørk, J., Boer, F. S. d., Johnsen, E. B., Schlatte, R., and Tarifa, S. L. T. (2013). User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 9(1):29–43.

[Black et al., 2002] Black, A. P., Carlsson, M., Jones, M. P., Kieburtz, R., and Nordlander, J. (2002). Timber: A programming language for real-time embedded systems. Technical report.

[Boer et al., 2007] Boer, F. S. d., Clarke, D., and Johnsen, E. B. (2007). A complete guide to the future. In *Programming Languages and Systems*, pages 316–330. Springer.

[Boer and Gouw, 2014] Boer, F. S. d. and Gouw, S. d. (2014). Combining Monitoring with Run-Time Assertion Checking. In *Formal Methods for Executable Software Models*, Lecture Notes in Computer Science, pages 217–262. Springer, Cham.

[Boer et al., 2013] Boer, F. S. d., Gouw, S. d., and Wong, P. Y. H. (2013). Run-Time Verification of Coboxes. In *Software Engineering and Formal Methods*, Lecture Notes in Computer Science, pages 259–273. Springer, Berlin, Heidelberg.

[Brandauer et al., 2015] Brandauer, S., Castegren, E., Clarke, D., Fernandez-Reyes, K., Johnsen, E. B., Pun, K. I., Tarifa, S. L. T., Wrigstad, T., and Yang, A. M. (2015). Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore. In *Formal Methods for Multicore Programming*, Lecture Notes in Computer Science, pages 1–56. Springer, Cham.

[Calheiros et al., 2011] Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A. F., and Buyya, R. (2011). CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50.

[Claessen and Hughes, 2011] Claessen, K. and Hughes, J. (2011). QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.*, 46(4):53–64.

[Clarke et al., 2010] Clarke, D., Helvensteijn, M., and Schaefer, I. (2010). Abstract Delta Modeling. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 13–22, New York, NY, USA. ACM.

[Clebsch et al., 2015] Clebsch, S., Drossopoulou, S., Blessing, S., and McNeil, A. (2015). Deny Capabilities for Safe, Fast Actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2015, pages 1–12, New York, NY, USA. ACM.

[Dean and Ghemawat, 2008] Dean, J. and Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107.

[Din et al., 2015] Din, C. C., Bubel, R., and Hähnle, R. (2015). KeY-ABS: A Deductive Verification Tool for the Concurrent Modelling Language ABS. In *Automated Deduction - CADE-25*, Lecture Notes in Computer Science, pages 517–526. Springer, Cham.

[Din et al., 2017] Din, C. C., Owe, O., and Bubel, R. (2017). Runtime Assertion Checking and Theorem Proving for Concurrent and Distributed Systems. pages 480–487.

[Doménech et al., 2017] Doménech, J., Genaim, S., Johnsen, E. B., and Schlatte, R. (2017). EasyInterface: A Toolkit for Rapid Development of GUIs for Research Prototype Tools. In *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 379–383. Springer, Berlin, Heidelberg.

[Eisenberg, 2015] Eisenberg, R. (2015). System FC, as implemented in GHC.

[Epstein et al., 2011] Epstein, J., Black, A. P., and Peyton-Jones, S. (2011). Towards Haskell in the cloud. In *ACM SIGPLAN Notices*, volume 46, pages 118–129. ACM.

[Flanagan and Felleisen, 1995] Flanagan, C. and Felleisen, M. (1995). The semantics of future and its use in program optimization. pages 209–220. ACM Press.

[Giachino et al., 2016a] Giachino, E., Gouw, S. d., Laneve, C., and Nobakht, B. (2016a). Statically and Dynamically Verifiable SLA Metrics. In *Theory and Practice of Formal Methods*, Lecture Notes in Computer Science, pages 211–225. Springer, Cham.

[Giachino et al., 2014] Giachino, E., Kobayashi, N., and Laneve, C. (2014). Deadlock Analysis of Unbounded Process Networks. In *CONCUR 2014 – Concurrency Theory*, volume 8704, pages 63–77. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Giachino et al., 2016b] Giachino, E., Laneve, C., and Lienhardt, M. (2016b). A framework for deadlock detection in core ABS. *Software & Systems Modeling*, 15(4):1013–1048.

[Gibbons, 2007] Gibbons, J. (2007). Datatype-Generic Programming. In *Datatype-Generic Programming*, Lecture Notes in Computer Science, pages 1–71. Springer, Berlin, Heidelberg.

[Göri et al., 2014] Göri, G., Johnsen, E. B., Schlatte, R., and Stolz, V. (2014). Erlang-Style Error Recovery for Concurrent Objects with Cooperative Scheduling. In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, Lecture Notes in Computer Science, pages 5–21. Springer, Berlin, Heidelberg.

[Gouw et al., 2016] Gouw, S. d., Mauro, J., Nobakht, B., and Zavattaro, G. (2016). Declarative Elasticity in ABS. In *Service-Oriented and Cloud Computing*, Lecture Notes in Computer Science, pages 118–134. Springer, Cham.

[Hewitt et al., 1973] Hewitt, C., Bishop, P., and Steiger, R. (1973). A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Holzmann, 2003] Holzmann, G. (2003). *Spin Model Checker, the: Primer and Reference Manual.* Addison-Wesley Professional, first edition.

[Imam and Sarkar, 2014] Imam, S. M. and Sarkar, V. (2014). Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, AGERE! '14, pages 67–80, New York, NY, USA. ACM.

[Jefferson and Sowizral, 1985] Jefferson, D. and Sowizral, H. (1985). Fast concurrent simulation using the Time Warp mechanism. In *SCS Conf. Distributed Simulation*, pages 63–69.

[Johnsen et al., 2010a] Johnsen, E. B., Hähnle, R., Schäfer, J., Schlatte, R., and Steffen, M. (2010a). ABS: A Core Language for Abstract Behavioral Specification. In *Formal Methods for Components and Objects*, Lecture Notes in Computer Science, pages 142–164. Springer, Berlin, Heidelberg.

[Johnsen et al., 2010b] Johnsen, E. B., Owe, O., Schlatte, R., and Tarifa, S. L. T. (2010b). Validating Timed Models of Deployment Components with Parametric Concurrency. In *Formal Verification of Object-Oriented Software*, Lecture Notes in Computer Science, pages 46–60. Springer, Berlin, Heidelberg.

[Johnsen et al., 2006] Johnsen, E. B., Owe, O., and Yu, I. C. (2006). Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1):23–66.

[Johnsen et al., 2012] Johnsen, E. B., Schlatte, R., and Tarifa, S. L. T. (2012). Modeling Resource-Aware Virtualized Applications for the Cloud in Real-Time ABS. In *Formal Methods and Software Engineering*, Lecture Notes in Computer Science, pages 71–86. Springer, Berlin, Heidelberg.

[Kiselyov and Laemmel, 2005] Kiselyov, O. and Laemmel, R. (2005). Haskell's overlooked object system. *arXiv:cs/0509027*. arXiv: cs/0509027.

[Kiselyov et al., 2004] Kiselyov, O., Lämmel, R., and Schupke, K. (2004). Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM.

[Kliazovich et al., 2010] Kliazovich, D., Bouvry, P., Audzevich, Y., and Khan, S. U. (2010). GreenCloud: A Packet-Level Simulator of Energy-Aware Cloud Computing Data Centers. pages 1–5. IEEE.

[Knuth, 1973] Knuth, D. E. (1973). *The art of computer programming*. Addison-Wesley series in computer science and information processing. Addison-Wesley Pub. Co, Reading, Mass.

[Lanese et al., 2014] Lanese, I., Lienhardt, M., Bravetti, M., Johnsen, E. B., Schlatte, R., Stolz, V., and Zavattaro, G. (2014). Fault Model Design Space for Cooperative Concurrency. In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, Lecture Notes in Computer Science, pages 22–36. Springer, Berlin, Heidelberg.

[Long et al., 2005] Long, Q., Liu, Z., Li, X., and Jifeng, H. (2005). Consistent code generation from UML models. In *2005 Australian Software Engineering Conference*, pages 23–30.

[Magalhães et al., 2010] Magalhães, J. P., Dijkstra, A., Jeuring, J., and Löh, A. (2010). A Generic Deriving Mechanism for Haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pages 37–48, New York, NY, USA. ACM.

[McBride, 2000] McBride, C. (2000). Dependently Typed Functional Programs and their Proofs.

[Misra, 1986] Misra, J. (1986). Distributed discrete-event simulation. *ACM Computing Surveys (CSUR)*, 18(1):39–65.

[Moreira et al., 2010] Moreira, T. G., Wehrmeister, M. A., Pereira, C. E., Pétin, J. F., and Levrat, E. (2010). Automatic code generation for embedded systems: From UML specifications to VHDL code. In *2010 8th IEEE International Conference on Industrial Informatics*, pages 1085–1090.

[Nakata and Saar, 2013] Nakata, K. and Saar, A. (2013). Compiling Cooperative Task Management to Continuations. In *Fundamentals of Software Engineering*, pages 95–110. Springer.

[Nipkow et al., 2002] Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). *Isabelle/HOL: a proof assistant for higher-order logic.* Number 2283 in Lecture notes in computer science. Springer, Berlin ; New York.

[Noll, 2001] Noll, T. (2001). A Rewriting Logic Implementation of Erlang. *Electronic Notes in Theoretical Computer Science*, 44(2):206–224.

[Nordlander, 2002] Nordlander, J. (2002). Polymorphic subtyping in O'Haskell. *Science of Computer Programming*, 43(2–3):93–127.

[Núñez et al., 2012] Núñez, A., Vázquez-Poletti, J. L., Caminero, A. C., Castañé, G. G., Carretero, J., and Llorente, I. M. (2012). iCanCloud: A Flexible and Scalable Cloud Infrastructure Simulator. *Journal of Grid Computing*, 10(1):185–209.

[Palacios et al., 2015] Palacios, A., Vidal, G., and Herbstritt, M. (2015). Towards Modelling Actor-Based Concurrency in Term Rewriting. Technical report, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany.

[Peyton Jones, 2003] Peyton Jones, S. L., editor (2003). *Haskell 98 language and libraries: the revised report.* Cambridge University Press, Cambridge, U.K. ; New York. OCLC: ocm51271691.

[Schäfer and Poetzsch-Heffter, 2010] Schäfer, J. and Poetzsch-Heffter, A. (2010). JCoBox: Generalizing Active Objects to Concurrent Components. In *ECOOP 2010 – Object-Oriented Programming*, Lecture Notes in Computer Science, pages 275–299. Springer, Berlin, Heidelberg.

[Sirjani et al., 2004] Sirjani, M., Movaghar, A., Shali, A., Boer, D., and S, F. (2004). Modeling and Verification of Reactive Systems using Rebeca. *Fundamenta Informaticae*, 63(4):385–410.

[Srinivasan and Mycroft, 2008] Srinivasan, S. and Mycroft, A. (2008). Kilim: Isolation-Typed Actors for Java. In *ECOOP 2008 – Object-Oriented Programming*, Lecture Notes in Computer Science, pages 104–128. Springer, Berlin, Heidelberg.

[Sulzmann et al., 2007] Sulzmann, M., Chakravarty, M. M., Jones, S. P., and Donnelly, K. (2007). System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66. ACM.

[Tarau, 2011] Tarau, P. (2011). Coordination and Concurrency in Multi-engine Prolog. In *Coordination Models and Languages*, Lecture Notes in Computer Science, pages 157–171. Springer, Berlin, Heidelberg.

[Vidal, 2014] Vidal, G. (2014). Towards Erlang Verification by Term Rewriting. In Gupta, G. and Peña, R., editors, *Logic-Based Program Synthesis and Transformation*, volume 8901, pages 109–126. Springer International Publishing, Cham.

[Walker and Runciman, 2015] Walker, M. and Runciman, C. (2015). Déjà fu: a concurrency testing library for haskell. pages 141–152. ACM Press.

[Wong et al., 2012] Wong, P. Y., Albert, E., Muschevici, R., Proença, J., Schäfer, J., and Schlatte, R. (2012). The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *International Journal on Software Tools for Technology Transfer*, 14(5):567–588.

[Wong et al., 2015] Wong, P. Y. H., Bubel, R., Boer, F. S. d., Gómez-Zamalloa, M., Gouw, S. d., Hähnle, R., Meinke, K., and Sindhu, M. A. (2015). Testing abstract behavioral specifications. *International Journal on Software Tools for Technology Transfer*, 17(1):107–119.