Fundamenta Informaticae 177 (2020) 203–234
DOI 10.3233/FI-2020-1988
IOS Press

A Formal, Resource Consumption-Preserving Translation from Actors with Cooperative Scheduling to Haskell*

Elvira Albert†
Universidad Complutense de Madrid, Madrid, Spain
elvira@sip.ucm.es

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

Enrique Martin-Martin
Universidad Complutense de Madrid, Madrid, Spain
emartinm@ucm.es

Abstract. We present a formal translation of a resource-aware extension of the Abstract Behavioral Specification (ABS) language to the functional language Haskell. ABS is an actor-based language tailored to the modeling of distributed systems. It combines asynchronous method calls with a suspend and resume mode of execution of the method invocations. To cater for the resulting cooperative scheduling of the method invocations of an actor, the translation exploits for the compilation of ABS methods Haskell functions with continuations.

The main result of this article is a correctness proof of the translation by means of a simulation relation between a formal semantics of the source language and a high-level operational semantics of the target language, i.e., a subset of Haskell. We further prove that the resource consumption of an ABS program extended with a cost model is preserved over this translation, as we establish an equivalence of the cost of executing the ABS program and its corresponding Haskell-translation. Concretely, the resources consumed by the original ABS program and those consumed by the

*This work was funded partially by the Spanish MICINN/FEDER, UE project RTI2018-094403-B-C31, the MINECO project TIN2015-69175-C4-2-R, and by the CM project S2018/TCS-4314.
†Address for correspondence: Fac. Informática UCM, C/ Profesor José García Santesmases 9, 28040 Madrid, Spain

Haskell program are the same, considering a cost model. Consequently, the resource bounds automatically inferred for ABS programs extended with a cost model, using resource analysis tools, are sound resource bounds also for the translated Haskell programs. Our experimental evaluation confirms the resource preservation over a set of benchmarks featuring different asymptotic costs.

Keywords: actor model, futures, cooperative multitasking, coroutine, continuation, functional programming, operational semantics

1. Introduction

An important class of applications requires some explicit control of the underlying hardware resources, like Cloud applications and sensor networks which involve high data rates and extensive in-network processing [1]. Depending on the abstraction level, resource-aware programming can be error-prone. A major challenge in resource-aware programming therefore is to provide such resource control at an abstraction level which allows for the application of tool-supported rigorously defined resource analysis.

Abstract Behavioural Specification (ABS) [2] 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 [3, 4] 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 [5]. 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.

ABS further supports resource-aware programming that has been successfully applied to, for example, modeling the resource management of an industrial case study for cloud architectures [6]. The resource-aware extension of ABS is based on a *cost model* that assigns a cost to each ABS instruction. Resource analysis [7] allows the automatic generation of resource bounds inferred for ABS programs extended with a cost model.

Whereas ABS has successfully been used to model [8], analyze [9], and verify [2] actor programs, the development of an ABS implementation for the "real" execution of such programs has been a major challenge. This is mainly due to the problem of implementing cooperative scheduling in an efficient manner (common languages as Java and C++ have to resort to instrumentation techniques, e.g. fibers [10]). ABS currently supports various official backends with different cooperative scheduling implementations:[1]

[1]See `https://abs-models.org/manual/\#-abs-backends` for more information about ABS backends.

- ABS→Maude is an interpreter based on term rewriting,
- ABS→Java uses heavyweight threads and manual stack management,
- ABS→Erlang uses lightweight threads and thread parking,
- ABS→Haskell uses lightweight threads and continuations.

The major problem addressed in this article is how we can formally show that the results of the resource analysis of ABS programs extended with a cost model also hold for the translated ABS programs that are executed. In this article, we focus on the Haskell translation because it provides two interesting features. First, the use of continuations in implementing cooperative scheduling allows for a close correspondence with the Structural Operational Semantics of ABS [2], which simplifies the theoretical results. On the other hand, Haskell is one of the fastest ABS backends. The rest of ABS backends are less suited for this task: it is easy to reason about Maude programs, but their execution is significantly slower; whereas Java and Erlang programs are as fast as Haskell ones but reasoning about them is harder.

1.1. Summary of contributions

The overall contribution of this article is a formal, resource-consumption preserving translation of the concurrency subset of the ABS language into Haskell, given as an adaptation of the canonical ABS→Haskell backend [11]. We opted for the Haskell backend relying on the hypothesis that Haskell serves as a better middleground between execution speed and most importantly semantic correctness. The translation is based on compiling ABS methods into Haskell functions with *continuations*—similar transformations have been performed in the actor-based Erlang language wrt. rewriting systems [12, 13] and rewriting logic [14], and in the translations of ABS to Prolog [15] and a subset of ABS to Scala [16] (although there are not official ABS backends in Prolog or Scala). 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 wrt. 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.

This article is an extended version of the paper that appeared in the LOPSTR'16 proceedings [17]. Concretely, we have added complete proofs for all the theoretical results, as well as included the intermediate semantics and the translations between global configurations and Haskell data structures used in Section 4. We have also extended the related work in Section 6 and improved the experimental evaluation in Section 5 with programs showing a wider variety of asymptotic complexities.

1.2. Organization of the article

In Section 2 we specify the syntax of the source language, a subset of ABS, and detail its operational semantics. Section 3 describes our target language and defines the compilation process. This process translates the simplified ABS program into a Haskell program where the original statements are represented as data representing the abstract syntax tree, which is evaluated using an evaluation function (`eval`). We present the correctness and resource preservation results in Section 4. These theoretical results state that the final value obtained by executing the Haskell program is a correct final value of the original ABS program, and that the resources consumed during the execution of the Haskell program are the same resources consumed in the evaluation of the original ABS program (considering a cost model that measures instructions executed, objects created, etc.). In order to simplify the proof of these results, we use an intermediate semantics, which is also presented in Section 4. In Section 5 we show that the runtime environment does not introduce any significant real overhead when executing ABS instructions, and show empirically that the upper bounds obtained by the cost analysis are sound for the Haskell translated programs. Finally, Section 6 reviews related work and Section 7 contains the conclusions and future work.

2. Source language

Our language is based on ABS [2], 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.

In this article, we will consider a simplification of 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 the correctness of the cooperative concurrency. In other words, the ABS language is stripped of its functional core, local variables, object groups [18] and algebraic data types definition (although we assume the input programs are well-typed w.r.t the ABS type-system). Note that these simplifications are done only for the sake of conciseness and do not invalidate the results of our translation, as they could be easily added into the presented framework. Concretely, algebraic data types and functions based on pattern matching will behave as synchronous invocations in our framework. Supporting local variables simply requires extending our heap with a more complex structure or introduce that structure in our local configurations. Finally, object groups combine different actors inside

a component so that their execution only interleaves cooperatively, i.e., when an object finishes or awaits. In this article, we are implicitly assuming that every object is its own object group, but the same reasoning could be extended to objects groups with more than one object.

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

```
1  main() {
2    node1 = new;
3    node2 = new;
4    v1 = ...;
5    v2 = ...;
6    f1 = node1 ! map(v1);
7    f2 = node2 ! map(v2);
8    await f1;
9    await f2;
10   r1 = f1.get;
11   r2 = f2.get;
12   r = reduce(r1,r2);
13   return r;
14 }
15 map(v) {
16   ... }
17 reduce(v1,v2) {
18   ... }
```

Figure 1: (a) Syntax of the source language (b) Simplified MapReduce task in ABS

The formal syntax of the statements S of the subset is shown in Figure 1(a). Values in our subset are references (object or futures) and integer numbers; values can be stored in the 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 $f\texttt{:=}x\texttt{!}m(\bar{y})$ 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 $x\texttt{:=}E$ 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. A method declaration D maps a method's name and formal parameters to a statement S (method body) followed by a `return` z statement that returns the value of the attribute z (both in

synchronous and asynchronous method calls). Therefore, every method has exactly 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. Given a program, we will use the notation $m(\bar{w}) \mapsto S \in D$ to obtain an instance of the method declaration m using the fresh parameter names $\bar{w}$, so S will be the body of the method with the fresh parameters and including the final return statement.

The program of Figure 1(b) shows a basic version of a MapReduce task [19] 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` at Line 2 (L2 for short) and `node2` (at L3), and invokes asynchronously `map` with some values $\mathtt{v}_1$ and $\mathtt{v}_2$ (L6–L7). In MapReduce, all `map` invocations must finish before executing the *reduce* phase: therefore, the `await` instructions in L8–L9 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 L10-L11 obtain the results from the futures `f1` and `f2`. Although `get` statements block the 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, L12 contains a synchronous-method self call to `reduce` that combines the partial results from the *map* phase.

2.1. Operational semantics

In order to describe the operational semantics of the language defined above we first need to introduce some concepts and notation. First, we consider a set $IVar$ of attributes and $LVar$ of method parameters. The values considered in this article are in the *Int* set: integer constants and dynamically generated references to objects and futures.

Definition 2.1. (Set of assignments Σ)
A *set of assignments* Σ is a mapping $IVar \to Int$ from attributes to integer values. An empty set of assignments is denoted by ϵ.

Definition 2.2. (Substitutions τ and closures (S, l))
A *substitution* τ is a mapping from method parameters *LVar* to integers: $\tau \in LVar \to Int$. By $S\tau$ we denote the instantiation obtained from S by replacing each variable x in S by $\tau(x)$.

A *closure* (S, l) consists of a statement S obtained by replacing its free variables by actual values and a future reference l, represented by an integer, for storing the return value.

Definition 2.3. (Heap h)
A *heap* h is a triple $(count, h_1, h_2)$ consisting of an integer number *count* and *partial* mappings (with finite disjoint domains) $h_1 : Int \to \Sigma$ and $h_2 : Int \to Int_\perp$, where $Int_\perp = Int \cup \{\perp\}$ ($\perp$ is used to denote "undefined"). The number *count* 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, which is a set of assignments. Finally, the function h_2 specifies for each existing future reference, i.e., a number n such $h_2(n)$ is defined, its return value (absence of which is indicated by $\perp$).

In the sequel we will simply denote the first component of h by $h(count)$, write $h(n)(x)$ for $h_1(n)(x)$, and $h(n)$ for $h_2(n)$. Note that $h(n)$ will always refer to the value of the future reference n in h_2, not the set of assignments Σ associated to object n in h_1.

We consider the following operations for updating heaps:

- $h[count \mapsto n]$ generates a heap equal to h but with the counter set to n.
- $h[(n) \mapsto v]$ creates a heap equal to h but storing the value v in the future variable n.
- $h[(n)(x) \mapsto v]$ is similar to the previous one, but storing the value v in the local variable x in object n
- $h[(n) \mapsto \epsilon]$ is used to generate a heap equal to h extended with the empty set of assignments for the object n.

Definition 2.4. (Local and global configurations)
The *local configuration* of an object is denoted by the pair $\langle n : Q, h\rangle$, where n is the object reference, Q is a list of closures and h is a heap. We use "$\cdot$" to concatenate lists of closures, 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 heap h.

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

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

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

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

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

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

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

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

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

Figure 2: Operational semantics: Local rules

Using the previously defined notions we can present the relation that describes the local behavior of an object, that is shown in Figure 2. (omitting the usual rules for sequential composition, `if` and `while` statements because they are standard).

Note that the first closure of the list Q is the active process of the object, so the different rules process the first statement of this closure. When the active process finishes or releases the object in an `await` statement, the next process in the list will become active, following a FIFO policy. The rule (ASSIGN) modifies the heap storing the new value of variable x of object n. It uses the function $\texttt{getVal}(h, n, V)$ to evaluate an expression V involving integer constants and variables using the set of assignments stored in the heap h for the object n. The (NEW) rule stores a new object reference in variable x, increments the counter of objects references and inserts an empty mapping ϵ for the variables of the new object m. Rule (GET) can only be applied if the future is available, i.e., if its value is not $\bot$. 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 $\bot$ 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 the *write-back variable*. This marking is formalized in the $\widehat{\cdot}^s$ function, defined as follows (recall that `return` is the last statement of any method):

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

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

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

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

Figure 3: Operational semantics: Global rules

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

We remark that the operational semantics shown in Figure 2 and 3 is very similar to the foundational ABS semantics presented in [2], considering that every object is a *concurrent object group* (see [2] for further details). The main difference is the representation of configurations: in [2] 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 heap. Finally, our operational semantics considers a FIFO policy in the processes of an object, whereas [2] left the scheduling policy unspecified.

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. In other words, 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.

3.1. Coroutines and continuation passing style

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 data structure that captures what the remaining of the program does (after the point). As an example, consider the Haskell program in Figure 4(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*

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

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

Figure 4: (a) Example program in direct style and (b) translated to CPS

style [20, 21] (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 Figure 4(b), or (for efficiency reasons) to only the subset that relies on continuation support, e.g. only those functions that need to suspend/resume.

3.2. Compilation of ABS programs to Haskell with CPS

In our case, ABS programs are translated to Haskell with CPS applied only to statements and methods, but not (sub)expressions. Figure 5 shows the types and datatypes used to express the ABS programs in Haskell with CPS. 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, which will only later be interpreted at runtime (see Section 3.3): 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 Figure 5 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

```
type Method = [Ref] → Ref → Maybe Ref → Stm → Stm
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
data B = B :∧ B | B :∨ B | :¬ B | V :≡ V
data V = A Ref | P Ref | I Int
   | Add V V | Sub V V ...
```

Figure 5: The syntax and types of the target language. Continuations are wave-underlined. The program/process final result type is double-underlined

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 heap; 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 shown in Figure 1-a). Boolean expressions are only appearing as predicates to If and While and are inductively constructed by the datatype B, which represents reference and integer comparison.

$$
\begin{array}{rlrl}
{}^{s}[\![\texttt{skip}]\!]_{k,wb} &= \texttt{Skip}\ k & {}^{s}[\![\texttt{x:=}V]\!]_{k,wb} &= \texttt{Assign}\ x\ {}^{V}[\![V]\!]\ k \\
{}^{s}[\![\texttt{await f}]\!]_{k,wb} &= \texttt{Await}\ f\ k & {}^{s}[\![\texttt{x:=new}]\!]_{k,wb} &= \texttt{Assign}\ x\ \texttt{New}\ k \\
{}^{s}[\![\texttt{return x}]\!]_{k,wb} &= \texttt{Return}\ x\ wb\ k & {}^{s}[\![\texttt{x:=f.get}]\!]_{k,wb} &= \texttt{Assign}\ x\ (\texttt{Get}\ f)\ k \\
{}^{s}[\![\texttt{return}^{*}\ \texttt{x}]\!]_{k,wb} &= \texttt{Return}\ x\ \texttt{Nothing}\ k & {}^{s}[\![\texttt{x:=y!m(}\bar{z}\texttt{)}]\!]_{k,wb} &= \texttt{Assign}\ x\ (\texttt{Async}\ y\ m\ \bar{z})\ k \\
{}^{s}[\![\texttt{return}^{z}\ \texttt{x}]\!]_{k,wb} &= \texttt{Return}\ x\ (\texttt{Just}\ z)\ k & {}^{s}[\![\texttt{x:=m(}\bar{z}\texttt{)}]\!]_{k,wb} &= \texttt{Assign}\ x\ (\texttt{Sync}\ m\ \bar{z})\ k
\end{array}
$$

$$
\begin{array}{rl}
{}^{s}[\![S_1; S_2]\!]_{k,wb} &= {}^{s}[\![S_1]\!]_{k',wb} \text{ with } k' = {}^{s}[\![S_2]\!]_{k,wb} \\
{}^{s}[\![\texttt{if}\ B\ \{S_1\}\ \texttt{else}\ \{S_2\}]\!]_{k,wb} &= \texttt{If}\ {}^{B}[\![B]\!]\ (\backslash k' \rightarrow {}^{s}[\![S_1]\!]_{k',wb})\ (\backslash k' \rightarrow {}^{s}[\![S_2]\!]_{k',wb})\ k \\
{}^{s}[\![\texttt{while}\ B\ \{S\}]\!]_{k,wb} &= \texttt{While}\ {}^{B}[\![B]\!]\ (\backslash k' \rightarrow {}^{s}[\![S]\!]_{k',wb})\ k
\end{array}
$$

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

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 6: Translation of ABS-subset programs to Haskell AST

The compilation of statements is shown in Figure 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 $k = \texttt{undefined}$ and $wb = \texttt{Nothing}$ 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, as defined by ${}^{m}[\![m]\!]$, that includes the compiled body. Figure 7 contains the translation to Haskell with CPS of the original MapReduce program written in ABS, whose source code was shown in Figure 1-b).

```
1  main, map, reduce :: Method
2  main [] this wb k =
3    Assign node1 New $
4    Assign node2 New $
5    Assign f1 (Async node1 map [v1])$
6    Assign f2 (Async node2 map [v2])$
7    Await f1 $
8    Await f2 $
9    Assign r1 (Get f1) $
10   Assign r2 (Get f2) $
11   Assign r (Sync reduce [r1,r2]) $
12   Return r wb k
13
14 map [v] this wb k = ...
15 reduce [a,b] this wb k = ...
16
17 -- Position in the attribute array
18 [node1,node2,f1,f2,r1,r2,r] = [0..]
```

Figure 7: The Haskell-translated running example of MapReduce

3.3. Runtime execution

The translation of an ABS program is a Haskell program with CPS where the statements have been encoded as data structures. Therefore, to execute the Haskell program we need a mechanism that processes these data structures until they produce a final result. This evaluation mechanism will be performed by `eval`, a Haskell function that takes a heap and an active object an executes a single statement.

For the `eval` function, we will use a heap representation in Haskell very similar to the heap notion presented in Definition 2.3. Concretely, the program heap is implemented as the triple: array of objects, array of futures and an `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., the type of the futures-array is `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 L18 of Figure 7).

The `eval` function accepts a `this` object reference, the current heap `h`, and the maximum number of attributes in objects `attrArrSize`, and executes a single statement of the head process in the process queue. It returns the executed statement, a new heap and those objects that have become active after the execution, i.e., `eval this heap :: IO (Stm, 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 other statements will keep the current process at the head of the process queue. A `Return` executed statement originating from an asynchronous call is responsible for re-activating the objects that are blocked on its resolved future. Figure 8 contains a snippet of the `eval` function, concretely the code that creates a new object, executes an unblocked await, and invokes a method asynchronously (the complete code can be found in the file *Eval.hs* from the repository `https://github.com/abstools/abs-haskell-formal/blob/master/src`). The first lines (L2–L6) extract the information `(attrs,pqueue)` of object `this` from the heap, selects the first process from `pqueue` and selects its first continuation `res`. The fragment L8–14 handles the creation of a

```
1  eval this h attrArrSize = do
2      (attrs ,pqueue) <- objects h `V.read` this
3      case S.viewl pqueue of
4          S.EmptyL -> error "scheduled an empty-proc object"
5          (Proc (destiny, c) S.:< restProcs) -> let res = c in case res of
6
7                 (...)
8               Assign lhs New k' -> do
9                     (attrs `V.write` lhs) $ newRef h
10                    updateObj $ Left k'
11                    initAttrVec <- V.replicate attrArrSize (-1)
12                    (objects h `V.write` newRef h) (initAttrVec, S.empty)
13                    h' <- incCounterMaybeGrow
14                    return (res, [this], h')
15              (...)
16             Await attr k' -> do
17                  fut <- V.read (futures h) =<< (attrs `V.read` attr)
18                  case fut of
19                       Left _ -> do -- unresolved future
20                              (...)
21                       Right _ -> do -- already-resolved
22                       future
23                             updateObj $ Left k'
24                           $ return (res, [this], h)
25              (...)
26             Assign lhs (Async obj m params) k' -> do
27                 calleeObj <- attrs `V.read` obj -- read the callee object
28                 (calleeAttrs, calleeProcQueue) <- (objects h `V.read` calleeObj)
29                 derefed_params <- mapM (attrs `V.read`) params -- read the passed attrs
30                 let newCont = m derefed_params calleeObj Nothing (error "...")
31                 let newProc = Proc (newRef h, newCont)
32                 (attrs `V.write` lhs) (newRef h)
33                 updateObj (Left k')
34                 (objects h `V.write` calleeObj) (calleeAttrs, calleeProcQueue S.|> newProc)
35                 (futures h `V.write` newRef h) (Left [ ]) -- create a new unresolved future
36                 h' <- incCounterMaybeGrow
37                 return (res, this :[calleeObj | S.null calleeProcQueue], h')
38     (...)
```

Figure 8: Snippet of the `eval` function.

new object, where `lhs` is the position in the vector `attrs` of the variable that will store the reference. In L9 the function updates the heap by storing a fresh reference (using the function `newRef`) in the variable `lhs`. Then, in L10, the function updates the process queue by pushing the next continuation `k'` in the front using function `updateObj`. In L11–12 the code creates an initial mapping `initAttrVec` for the new object and inserts it in the heap with an empty process queue (`S.empty` represents ϵ). Finally, it increments the reference counter using the function `incCounterMaybeGrow`[2] and returns `(res,[this],h')`. The fragment L16–24 handles `await` statements, omitting unresolved ones. Here `attr` is the position in the vector `attrs` of the future variable, so the code extracts its value from the heap (L17). If that value is defined (it has the form `Right _`) then the `await` is not blocked and proceeds by appending the continuation `k'` in the front of the process queue and returning `(res,[this],h)`—see L21–24). The fragment L26–37 handles asynchronous method calls. `lhs` and `obj` are the positions in the vector `attrs` of the variable that will store the future and the reference to the object that will execute the method, `m` is the Haskell function that is the translation of method m, `params` is the list of variables (the arguments of the method invocation), and `k'` is the continuation. 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`). Line 30 invoke `m` to obtain the continuation related to the asynchronous call. Line 32 stores the new reference `newRef h` in position `lhs`, and line 33 updates the heap by inserting the continuation `k'` in the front of the process queue of the current object o_n. The next two lines create and insert in the back of the process queue of object `obj` a new process with continuation `newCont` and destiny the new reference `newRef h`. L35 creates a new undefined future variable, i.e., with value `Left [ ]`, and L36 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. Note that if the callee object queue is empty then the list of objects returned will be `[this]`, otherwise it will be `[this,calleeObj]`.

On top of `eval`, 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.

4. Correctness and resource preservation

To prove that the translation is correct and resource preserving, we provide an intermediate semantics $\rightarrowtail$ for the Haskell translated programs. This semantics represents an intermediate layer of abstraction between the original ABS program and the target Haskell with CPS, and its main goal is simplifying the proofs. The $\rightarrowtail$ semantics is depicted in Figure 9, and 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 Figure 9 presents two main differences w.r.t. that in Figures 2 and 3 of Section 2. First, the list $[\overline{o_m}]$ is used to apply a *round-robin* policy: the first unblocked

[2]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.

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

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

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

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

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

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

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

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

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

Figure 9: Intermediate semantics.

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 methods `m` with the adequate parameters. Nevertheless, the rules of the $\rightarrowtail$ semantics correspond with the semantic rules in Section 2. When needed, we decorate a $\rightarrowtail$ step with the object reference o_n and the continuation *res* executed: $\rightarrowtail^{o_n}_{res}$.

Given a list $[\overline{o_m}]$ we use the notation $[\overline{o_{i\to k}}]$ for the sublist $[o_i, o_{i+1}, \ldots, o_k]$, 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 $[\overline{o_m}]$ if its process queue q_n is empty. In both cases, they advance the list of objects to o_{n+1}.

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

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

Our goal is proving that a sequence of `eval` steps represents a valid trace w.r.t. the semantics $\rightarrow$ in Section 2. We will proceed in two steps: First, we will prove that an `eval` step is also a valid step w.r.t. the intermediate semantics $\rightarrowtail$; and then we will prove that every $\rightarrowtail$-step can be performed with $\rightarrow$ using suitable configurations. The first lemma states that the results of invoking `eval` on the first unblocked object o_n are valid w.r.t. $\rightarrowtail$. To simplify notation, we consider an auxiliary function $\mathit{updL}([\overline{o_m}], o_n, l) = [\overline{o_{n+1\to m}}] : [\overline{o_{1\to n-1}}] : l$ that updates the list of object references.

Lemma 4.1. (Soundness of `eval` w.r.t. $\rightarrowtail$)
If the next unblocked object is $\mathit{nextObject}(h, [\overline{o_m}]) = o_n$ and `eval` o_n `h = (res,l,h')` then $(h, [\overline{o_m}]) \rightarrowtail^{o_n}_{res} (h', \mathit{updL}([\overline{o_m}], o_n, l))$.

Proof:
By case distinction on the portion of the `eval` code that processes `res` (see Figure 8). Here we only show 3 cases, but the rest are analogous. Notice that the current object `this` represents the object o_n.

- `res = Assign lhs New k'`
  Here, `lhs` is the position in the vector `attrs` of the variable `x`. The fresh reference extracted from the heap with `newRef` is $h(\mathit{count})$, and `S.empty` represents the empty queue ϵ. Then we

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

can perform the $\rightarrowtail$-step:

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

It is clear that $h' = $ `h'` because the updates are the same in both the code and the $\rightarrowtail$-step, and $[\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}] \equiv updL([\overline{o_m}], o_n, [o_n])$.

- `res = Await attr k'`
  `attr` is the position in the vector `attrs` of the future variable f, and as before $[\overline{o_{n+1 \to m}}] : [\overline{o_{1 \to n}}] \equiv updL([\overline{o_m}], o_n, [o_n])$. Therefore we can perform the following $\rightarrowtail$-step:

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

- `Assign lhs (Async obj m params) k'`
  `lhs` and `obj` are the positions of the future variable x and the reference to the callee object y (o_y) in the vector `attrs`. The continuation obtained from the invocation of `m` (`newCont` in the code, k'' in the $\rightarrowtail$-step) is combined with a new reference—`newRef h`, that returns the counter of references, and is the same as $h(count)$—to create a new process `newProc` that is inserted as the last element of the queue of the callee object (o_y). An undefined variable `Left []` is stored in reference $h(count)$, and that counter is incremented. Therefore we can perform the following $\rightarrowtail$-step:

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

It is easy to see that `h'` is equal to h' since they have received the same updates in variable x, counter, future variables and the process queues of o_y (the callee object) and o_n (the current object).

□

$$
\begin{array}{ll}
{}^{c}[\![\langle C,h\rangle]\!] = (h', act), \text{where} & {}^{q}[\![\epsilon]\!] \quad = \epsilon \\
\quad C = \{(o_1,Q_1),\ldots,(o_m,Q_m)\} \text{ and} & {}^{q}[\![(S,l)\cdot Q]\!] \quad = ({}^{s}[\![S]\!],l)\cdot {}^{q}[\![Q]\!] \\
\quad act = [o_n \mid (o_n,Q_n)\in C, Q_n \neq \epsilon] & \\
\quad h' = h[\overline{(o_i)(\mathcal{Q}) \mapsto {}^{q}[\![Q_i]\!]}] &
\end{array}
$$

Figure 10: Translation from source to target configurations.

The next step is to prove the soundness of $\rightarrowtail$ w.r.t. $\rightarrow$, i.e., that every $\rightarrowtail$-step can be performed using $\rightarrow$. In order to relate these two semantics, we first need a translation from runtime configurations $\langle C, h\rangle$ of Section 2 to concrete Haskell data structures used in the intermediate $\rightarrowtail$ semantics and vice versa. This translation is defined in Figure 10. 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. 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, object attributes and method parameters become Integers, and those $Int_\perp$ values stored 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.

Considering the translation ${}^{c}[\![\cdot]\!]^{-1}$ we can state the soundness of $\rightarrowtail$ as follows: a step $t_A \rightarrowtail^{o_n}_{S} t_B$ can be performed with $\rightarrow$ using the translated configurations. Concretely:

Lemma 4.2. (Soundness of $\rightarrowtail$ w.r.t. $\rightarrow$)
If $t_A \rightarrowtail^{o_n}_{S} t_B$ then ${}^{c}[\![t_A]\!]^{-1} \rightarrow^{o_n}_{S} {}^{c}[\![t_B]\!]^{-1}$.

Proof:
By case distinction on the rule applied to perform the step. The reasoning is similar in all the cases, so we only detail the steps (ASSIGN), (ASYNC), and (RETURN$_A$). For simplicity, we will identify the statement S used in the step $t_A \rightarrowtail^{o_n}_{S} t_B$ (which is a Haskell value of type `Stm`) with the corresponding statement using the syntax of Figure 1. For example, we identify `Assign x V k'` with `x:=V`.

- **(ASSIGN).**

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

The inverse translation of t_A is defined as

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

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

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

It is clear that ${}^{c}[\![t_B]\!]^{-1} = B$ because the set of object references in B is $\{\overline{o_m}\}$ and B contains $(o_n : {}^{q}[\![(k,l)\cdot q]\!]^{-1})$. Moreover, h' is the translation of h'_c because is the result of transforming h with the same operations as h_c, i.e., $h'(o_n)(\mathcal{Q}) = {}^{q}[\![(k,l)\cdot q]\!]^{-1} = (S,l)\cdot Q$ and $h'_c(o_n)(x) = v$.

- **(ASYNC).**

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

The inverse translation of t_A is defined as

$${}^{c}[\![t_A]\!]^{-1} = A = ((o_n : (\texttt{f:=x!m}(\bar{z}); S, l)\cdot Q_n) \cup (o_x : Q_x) \cup C, h)$$

where ${}^{s}[\![k]\!]^{-1} = S$, ${}^{q}[\![q_n]\!]^{-1} = Q_n$, ${}^{q}[\![q_x]\!]^{-1} = Q_x$, 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:

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

where the step inside object o_n uses rule (ASYNC)

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

The set of object references in B is $\{\overline{o_m}\}$, so it contains the same references as the list $s = \mathit{newQ}_{\mathit{add}}([\overline{o_m}], o_n, o_x)$. On the other hand, B contains the updated list of closures $(o_n : {}^{q}[\![(k,l)\cdot q_n]\!]^{-1})$ and $(o_x : {}^{q}[\![q_x \cdot (k', l')]\!]^{-1})$—notice that the translation of the continuation k' is

$S' = \widehat{(S_m\tau)}^*$. Finally, h' is the translation of h'_c because is the result of transforming h with the same operations as h_c, i.e., $h'(o_n)(f) = l'$, $h'(count = l' + 1)$, and $h'(l') = \perp$.

- **(RETURN$_A$).**

$$
(\text{RETURN}_A)\frac{\begin{array}{c} nextObject(h_c, [\overline{o_m}]) = o_n \\ h_c(o_n)(\mathcal{Q}) = (\texttt{Return}\; x\; \texttt{Nothing}\; k, l) \cdot q_n \\ newQ_{del}([\overline{o_m}], o_n, q_n) = s \\ h'_c = h_c[l \mapsto h_c(o_n)(x), (o_n)(\mathcal{Q}) \mapsto q_n] \end{array}}{t_A \equiv (h_c, [\overline{o_m}]) \rightarrowtail^{o_n}_{\texttt{return}^*\ \texttt{x}} (h'_c, s) \equiv t_B}
$$

The inverse translation of t_A is defined as

$$
{}^c[\![t_A]\!]^{-1} = A = (o_n : (\texttt{return}^*\ \ \texttt{x}; S, l) \cdot Q_n) \cup C, h)
$$

where ${}^s[\![k]\!]^{-1} = S$, ${}^q[\![q_n]\!]^{-1} = Q_n$, 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:

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

If $q_n = \epsilon$ then o_n will not appear in s, but Q_n will be empty as well. On the other hand, B contains the updated list of closures $(o_n : {}^q[\![q_n]\!]^{-1})$. Finally, h' is the translation of h'_c because is the result of transforming h with the same operations as h_c, i.e., $h'(l) = h_c(o_n)(c) = h(o_n)(x)$. □

Based on the previous lemmas, we can prove the soundness of the traces, i.e., every sequence of concatenated `eval` steps represents a valid trace w.r.t. $\to$. Given a heap h_i and an object o_i from the list of active objects in s_i (those whose queue is not empty), `eval` performs one step of execution processing the instruction $\texttt{res}_i$, generating a list of new active objects l_i and producing a new heap h_{i+1}. These `eval` invocations can be chained, producing a sequence of heaps and active objects that can be translated to global configurations that form a valid trace wrt. $\to$, and execute the same statement $\texttt{res}_i$ in the same object o_i. Therefore, the following result expresses the expected one-to-one relationship between the Haskell program evaluated using `eval` invocations and the original ABS program evaluated by the $\to$ semantics. 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.

Theorem 4.3. (Trace soundness)
Consider an initial state (h_1, s_1) and a sequence of $n - 1$ consecutive `eval` steps defined as:

1) $nextObject(h_i, s_i) = o_i$,

2) $\texttt{eval o}_i\,\texttt{h}_i = \texttt{(res}_i\texttt{, l}_i\texttt{, h}_{i+1}\texttt{)}$,

3) $s_{i+1} = \mathit{updL}(s_i, o_i, l_i)$.

Then there is a trace ${}^{c}[\![(h_1, s_1)]\!]^{-1} \rightarrow^{o_1}_{res_1} {}^{c}[\![(h_2, s_2)]\!]^{-1}_{c} \rightarrow^{o_2}_{res_2} \ldots \rightarrow^{o_{n-1}}_{res_{n-1}} {}^{c}[\![(h_n, s_n)]\!]^{-1}$.

Proof:
By induction on the number of `eval` steps using Lemmas 4.1 and 4.2. □

Note that it is not possible to obtain a similar result about trace completeness since the $\rightarrow$-semantics in Figure 3 selects the next object to execute nondeterministically (random scheduler), whereas the intermediate $\rightarrowtail$-semantics in Figure 9 follows a concrete *round-robin* scheduling policy. As a final remark, notice that the intermediate semantics $\rightarrowtail$ can be seen as a *specification* of the `eval` function. Therefore, in addition to be used as an intermediate layer to simplify proofs, the semantics $\rightarrowtail$ could also be used to guide the correctness proof of `eval` if we wanted to use semi-automatic proof assistance tools like *Isabelle* [22] or Coq [23], and also to generate tests automatically using *QuickCheck* [24].

4.1. Preservation of resource consumption and resource bounds

A strong feature of our translation is that the Haskell-translated program preserves the *resource consumption* of the original ABS program. As in [25] 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 then $\mathcal{M}(S) = 1$, i.e., each instruction has cost one. However, if one wants to measure memory consumption, we have that

$$\mathcal{M}(S) = \begin{cases} \mathtt{c} & \text{if } S \text{ is the statement } \mathtt{x := new} \\ 0 & \textit{i.o.c.} \end{cases}$$

where `c` refers to the size of an object reference. 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*. This notion is analogously adapted to traces of $\rightarrowtail$-steps and also to sequences of `eval` invocations because it only considers the object o_i involved and the statement S_i executed in every step. Since the trace soundness in Theorem 4.3 states that the `eval` function performs the same steps as some trace $\mathcal{T}$, the consumption preservation is a straightforward corollary:

Corollary 4.4. (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:

1) $o_i = nextObject(h_i, s_i)$,
2) $\texttt{eval o}_i\ \texttt{h}_i = (\texttt{res}_i\texttt{, l}_i\texttt{, h}_{i+1})$,
3) $s_{i+1} = updL(s_i, o_i, l_i)$.

Then there is a trace $\mathcal{T} = {}^c[\![(h_1, s_1)]\!]^{-1} \rightarrow^{o_1}_{res_1} {}^c[\![(h_2, s_2)]\!]^{-1}_c \rightarrow^{o_2}_{res_2} \ldots \rightarrow^{o_{n-1}}_{res_{n-1}} {}^c[\![(h_n, s_n)]\!]^{-1}$ such that $\mathcal{C}(\mathcal{T}_E, o, \mathcal{M}) = \mathcal{C}(\mathcal{T}, o, \mathcal{M})$.

Using Corollary 4.4 we can prove that the upper bounds on resource consumption that are inferred for the ABS programs using resource analyzers like [25] are also valid upper bounds for the Haskell translated code. We base our results on Theorem 3 from [25], which states that the cost of any $\leadsto$-trace $\mathcal{T}$ using the semantics $\leadsto$ in [25] is less than or equal to the obtained upper bound. The only step we need to prove is that for any $\rightarrow$-trace we have an equivalent $\leadsto$-trace with the same cost. These two semantics have some syntactic differences but they have the same behavior, so the correspondence is direct. In this case, the correspondence is not one-to-one because the semantics $\leadsto$ 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 $\leadsto$-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 calculi.

The language presented in Section 2 and its semantics in Figures 2 and 3 are a simplified version of those in [25]. The main differences are: 1) The representation of the states, 2) the syntax of method invocations (both synchronous and asynchronous), and 3) the consideration of local variables and class declarations. In [25] 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* related to object o (note that it is not a global 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 $fut(l, v)$, where l is the future identifier and v its value, possibly $\perp$. The operational semantics rewrites states $St \leadsto St'$.

For simplicity and for making the article self-contained, in Figure 11 we present the operational semantics $\leadsto$ of [25] adapted to our source language: attributes can be directly assigned by `new` and `get` instructions or arbitrary expressions in the right-hand side, future variables are attributes instead of local variables (note that we do not consider local variables), and there are not different classes for creating objects. Clearly, these changes do not affect the upper bounds or the results obtained in [25]. Rule (1 & 2) of Figure 11 evaluates the assignment of an attribute from an expression, so it behaves like rule (ASSIGN) of Figure 2. This rules combines rules (1) and (2) from [25], that handle attribute and local variable assignment respectively. Rule (3) of Figure 11 processes an object creation, where $newRef$ generates a new object identifier and $newHeap$ creates an initial heap for an object (in our setting all the objects are initialized with an empty heap ϵ). This rule is similar to rule (NEW) of Figure 2. Rules (4) and (5) handle synchronous and asynchronous calls respectively, and are similar to rules (SYNC) and (ASYNC) in Figure 2. In these rules, we use the notation D^x and D^* to denote the set of method definitions where all the `return` statements are decorated with the variable x or

$$(1\&2)\frac{v = eval_e(V,h)}{\{ob(n,_,h,\langle tv,\texttt{x:=}V;S\rangle,Q)|R\} \rightsquigarrow^n_{\texttt{x:=}V} \{ob(n,_,h[x\mapsto v],\langle tv,S\rangle,Q)|R\}}$$

$$(3)\frac{m = newRef() \quad newHeap(_,\epsilon)}{\{ob(n,_,h,\langle tv,\texttt{x:=new};S\rangle,Q)|R\} \rightsquigarrow^n_{\texttt{x:=new}} \{ob(n,_,h[x\mapsto m],\langle tv,S\rangle,Q),\{ob(m,_,\epsilon,\epsilon,\emptyset)|R\}}$$

$$(4)\frac{(p(\bar{w})\mapsto S_p)\in D^x \quad \tau=[\bar{w}\mapsto h(n)(\bar{z})]}{\{ob(n,_,h,\langle tv,\texttt{call(b,p(this,}\bar{z}\texttt{,x));}S\rangle,Q)|R\} \rightsquigarrow^n_{\texttt{m(}\bar{z}\texttt{)}} \{ob(n,_,h,\langle tv,S_p\tau;S\rangle,Q)|R\}}$$

$$(5)\frac{h(x)=d \quad (p(\bar{w})\mapsto S_p)\in D^* \quad \tau=[\bar{w}\mapsto h(\bar{z})] \quad l=newFut()}{\{ob(n,_,h,\langle tv,\texttt{call(m,p(this,}\bar{z}\texttt{,f));}S\rangle,Q),ob(d,_,h_d,\langle tv_d,S_d\rangle,Q_d)|R\} \rightsquigarrow^n_{\texttt{d.m(l,}\bar{z}\texttt{)}} \{ob(n,_,h[\texttt{f}\mapsto l],\langle tv,S\rangle,Q),ob(d,_,h_d,\langle tv_d,S_d\rangle,\{\langle[\texttt{ret}\mapsto l],S_p\tau\rangle\}\cup Q_d),fut(l,\bot)|R\}}$$

$$(6)\frac{v=h(x)}{\{ob(n,_,h,\langle tv,\texttt{return}^z\ \texttt{x};S\rangle,Q)|R\} \rightsquigarrow^n_{\texttt{return}^z\ \texttt{x}} \{ob(n,_,h[z\mapsto v],\langle tv,S\rangle,Q)|R\}}$$

$$(7)]\frac{v=h(x)}{\{ob(n,_,h,\langle[\texttt{ret}\mapsto l],\texttt{return}^*\ \texttt{x};\rangle,Q),fut(l,\bot)|R\} \rightsquigarrow^n_{\texttt{return}^*\ \texttt{x}} \{ob(n,_,h,\epsilon,Q),fut(l,v)|R\}}$$

$$(8)\frac{h(f)=l \quad v\neq\bot}{\{ob(n,_,h,\langle tv,\texttt{x:=f.get};S\rangle,Q),fut(l,v)|R\} \rightsquigarrow^n_{\texttt{x:=f.get}} \{ob(n,_,h[x\mapsto v],\langle tv,S\rangle,Q),fut(l,v)|R\}}$$

$$(9) h(f)=l \quad v\neq\bot \quad \{ob(n,_,h,\langle tv,\texttt{await f};S\rangle,Q),fut(l,v)|R\} \rightsquigarrow^n_{\texttt{await f}} \{ob(n,_,h,\langle tv,S\rangle,Q),fut(l,v)|R\}$$

$$(10)\frac{h(f)=l}{\{ob(n,_,h,\langle tv,\texttt{await f};S\rangle,Q),fut(l,\bot)|R\} \rightsquigarrow^n_{\texttt{await f}} \{ob(n,_,h,\epsilon,\langle tv,\texttt{await f};S\rangle\cup Q,fut(l,\bot)|R\}}$$

$$(11)\frac{\langle tv,S\rangle\in Q}{\{ob(n,_,h,\epsilon,Q)|R\} \rightsquigarrow^n_\epsilon \{ob(n,_,h,\langle tv,S\rangle,Q\setminus\{\langle tv,S\rangle\})|R\}}$$

Figure 11: Operational Semantics of $\rightsquigarrow$ from [25] adapted to our source language.

the mark $*$ respectively. Note that the future that will store the result of the asynchronous call in rule (5) is stored in the table of local variables, related to the special symbol `ret`. Since our source language does not consider local variables, this symbol will be the only one appearing in the table of local variables tv of any closure. Rules (6) and (7) process `return` statements for synchronous and asynchronous calls, and behave like rules (RETURN$_S$) and (RETURN$_A$) in Figure 2. Rule (8) handles a `get` statement like rule (GET). Rules (9) and (10) process `await` statements similarly to

rules (AWAIT I) and (AWAIT II). Finally, rule (11) nondeterministically selects one closure from the object's queue when the current object is idle. This rule does not have any corresponding rule in the $\rightarrow$ semantics in Figure 2 because that semantics automatically selects the next closure of the queue when the current closure finishes—with rule (RETURN$_A$)—or becomes blocked in an `await` statement—with rule (AWAIT II). However, that fixed selection criterion of $\rightarrow$ is a valid choice in $\rightsquigarrow$, which is more general.

$$\begin{array}{rcl} \|\langle C,h\rangle\| & = & \{ob(n,_,h_1(n),a,t) \mid (n:Q)\in C, (a,t)=\|Q\|_q\} \cup \\ & & \{ob(o,_,\epsilon,\epsilon,\emptyset) \mid o\in dom(h_1)\setminus objs(C)\} \cup \\ & & \{fut(l,v) \mid l\in dom(h_2), h(l)=v\} \end{array}$$

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

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

where $h=(count,h_1,h_2)$, $\|V\|_v$ is the straightforward translation of variables, references and integer expressions; $objs(C)$ returns the set of object identifiers in the set C, and $dom(h_i)$ returns the domain of the heap h_i.

Figure 12: Translation from configurations $\langle C,h\rangle$ to states St in [25]

Before proving the equivalence between $\rightarrow$-traces and $\rightsquigarrow$-traces we need a translation of configurations $\langle C,h\rangle$ and states St. This translation, denoted as $\|\langle C,h\rangle\|$, is detailed in Figure 12. The main ideas of the translation are:

- Every object n in the configuration C creates an $ob()$ quintuple where the object heap is $h_1(n)$.
- Every object o created that has not been executed yet (i.e., it appears in $dom(h_1)$ but not in $objs(C)$) creates an empty $ob()$ quintuple.
- Futures in h_2 are represented as $fut(l,v)$ pairs.
- Closures are translated straightforwardly, and the future is inserted in the table of local variables associated to the symbol `ret`.
- Closures queue is translated as a set of translated closures.

Moreover, this translation can be composed with ${}^c[\![\cdot]\!]^{-1}$ to translate Haskell configurations (h,s) into states St, namely $\langle\!\langle(h,s)\rangle\!\rangle = \|{}^c[\![(h,s)]\!]^{-1}\|$. We also define the notion of *relevant* traces of $\rightsquigarrow$ steps, i.e., those that execute an actual statement.

Definition 4.5. (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 a statement (i.e., its decoration is different from ϵ):

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

With these notions we can now define the soundness of $\rightarrow$ w.r.t. $\rightsquigarrow$, i.e., any $\rightarrow$-step can be expressed by a $\rightsquigarrow$-step executing the same statement in the same object, or by two $\rightsquigarrow$-steps where the first one executes the same statement in the same object and the second does not execute any instruction.

Lemma 4.6. (Soundness of $\rightarrow$ w.r.t. $\rightsquigarrow$)
If $\langle C, h\rangle \rightarrow^o_S \langle C', h'\rangle$ then $\|\langle C, h\rangle\| \rightsquigarrow^o_S \|\langle C', h'\rangle\|$ or $\|\langle C, h\rangle\| \rightsquigarrow^o_S St \rightsquigarrow^o_\epsilon \|\langle C', h'\rangle\|$

Proof:
By case distinction on the derivation applied to perform the $\rightarrow$-step. We present only the cases for assignment, blocked await (the only case that requires two $\rightsquigarrow$ steps) and synchronous calls; the remaining cases are analogous.

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

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

The translation of A is

$$\|A\| = \{ob(n, _, h_1(n), \langle [\texttt{ret} \mapsto l], \texttt{x:=}\|V\|_V; \|S\|_s\rangle, Q_{tr})|R\}$$

where R is the rest of objects and future variables not involved in the step and Q_{tr} the translation of Q. From $\|A\|$ it is possible to perform a $\rightsquigarrow$-step using rule (1 & 2) in Figure 11, reaching $\|B\|$:

$$(1\&2)\frac{v = eval_e(\|V\|_V, h_1(n))}{\begin{array}{l}\|A\| \equiv \{ob(n, _, h_1(n), \langle [\texttt{ret} \mapsto l], \texttt{x:=}\|V\|_V; \|S\|_s\rangle, Q_{tr})|R\} \rightsquigarrow^n_{\texttt{x:=}\|V\|_V} \\ \{ob(n, _, h_1(n)[x \mapsto v], \langle [\texttt{ret} \mapsto l], \|S\|_s\rangle, Q_{tr})|R\} \equiv \|B\|\end{array}}$$

Note that $eval_e$ is the function in [25] that computes the value of simple right-hand sides of assignments, so it behaves exactly like $\texttt{getVal}(h, n, V)$.

- **(Internal)+(Await II)**. This case involves 2 $\rightsquigarrow$-steps: one that evaluates an `await f` that cannot continue and releases the object, and one that schedules the next task in the object.

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

where $h(n)(f) = l$. 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:

$$\|A\| = \{ob(n, _, h_1(n), \langle[\texttt{ret} \mapsto l], \texttt{await f}; \|S\|_s\rangle, Q_{tr}), fut(l, \bot)|R\}$$

From $\|A\|$ we can perform a $\leadsto$-step using rule (10) from Figure 11:

$$(10)\frac{h(n)(f) = l}{\begin{array}{l}\|A\| \equiv \{ob(n, _, h_1(n), \langle[\texttt{ret} \mapsto l], \texttt{await f}; \|S\|_s\rangle, Q_{tr}), fut(l, \bot)|R\} \leadsto^n_{\texttt{await f}} \\ \{ob(n, _, h_1(n), \epsilon, \langle[\texttt{ret} \mapsto l], \texttt{await f}; \|S\|_s\rangle \cup Q_{tr}), fut(l, \bot)|R\} = A'\end{array}}$$

Then from the state A' we can apply rule (11) to schedule the first task a in the queue:

$$(11)\frac{a \in \langle[\texttt{ret} \mapsto l], \texttt{await f}; \|S\|_s\rangle \cup Q_{tr}}{\begin{array}{l}A' \equiv \{ob(n, _, h_1(n), \epsilon, \langle[\texttt{ret} \mapsto l], \texttt{await f}; \|S\|_s\rangle \cup Q_{tr}), fut(l, \bot)|R\} \leadsto^n_\epsilon \\ \{ob(n, _, h_1(n), a, t), fut(l, \bot)|R\} = \|B\|\end{array}}$$

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

- **(INTERNAL)+(SYNC).**

$$(\text{INTERNAL})\frac{(\text{SYNC})\dfrac{(m(\bar{w}) \mapsto S_m) \in D \quad \tau = [\bar{w} \mapsto h(n)(\bar{z})] \quad S' = \widehat{(S_m\tau)}^x}{\langle n{:}(\texttt{x:=m(}\bar{z}\texttt{)}; S, l) \cdot Q, h\rangle \to \langle n{:}(S'; S, l) \cdot Q, h\rangle}}{A \equiv \langle (n : (\texttt{x:=m(}\bar{z}\texttt{)}; S, l) \cdot Q) \cup C, h\rangle \to^n_{\texttt{x:=m(}\bar{z}\texttt{)}} \langle (n : (S'; S, l) \cdot Q) \cup C, h\rangle \equiv B}$$

The translation of A is

$$\|A\| = \{ob(n, _, h_1(n), \langle[\texttt{ret} \mapsto l], \texttt{call(b,m(this,}\bar{z}\texttt{,_))}; \|S\|_s\rangle, Q_{tr})|R\}$$

where R is the rest of objects and future variables not involved in the step and Q_{tr} the translation of Q. From $\|A\|$ it is possible to perform a $\leadsto$-step using rule (4) in Figure 11, reaching $\|B\|$:

$$(4)\frac{(m(\bar{w}) \mapsto S_m) \in \|D\|^x \quad \tau = [\bar{w} \mapsto h(n)(\bar{z})]}{\begin{array}{l}\|A\| \equiv \{ob(n, _, h_1(n), \langle[\texttt{ret} \mapsto l], \texttt{call(b,m(this,}\bar{z}\texttt{,_))}; \|S\|_s\rangle, Q_{tr})|R\} \leadsto^n_{\texttt{m(}\bar{z}\texttt{)}} \\ \{ob(n, _, h_1(n), \langle[\texttt{ret} \mapsto l], S_m\tau; \|S\|_s\rangle, Q_{tr})|R\} \equiv \|B\|\end{array}}$$

$\|D\|^x_{sync}$ 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 attribute x. We consider a simplification of the operational semantics in [25] where synchronous methods return exactly one value, thus the last instruction of a synchronous method stores the final value in the corresponding attribute. In this case is easy to check that $\|\widehat{(S_m\tau)}^x\| = S_m\tau$. □

The previous lemma can be extended to traces considering only the relevant steps that execute actual statements (i.e., ignoring $\leadsto$ steps decorated with ϵ):

Lemma 4.7. 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 $\mathcal{T}_C = \|A_1\| \leadsto^* \|A_n\|$ such that $rel(\mathcal{T}_C) = \|A_1\| \leadsto^{o_1}_{S_1} \|A_2\| \leadsto^{o_2}_{S_2} \ldots \leadsto^{o_{n-1}}_{S_{n-1}} \|A_n\|$.

Proof:
Straightforward by induction on the number of steps in the trace $\mathcal{T}$, and applying Lemma 4.6. □

Finally, we can state the soundness of the cost upper bound w.r.t. a sequence of `eval` steps. Considering a cost model $\mathcal{M}$, we denote by $UB^o_{\mathcal{M}}$ the upper bound on the cost of any execution starting from the `main` method restricted to object o. Therefore the soundness of upper bounds is stated as:

Theorem 4.8. (Upper bound soundness)
Let P be a program and $\mathcal{T}_E$ a sequence of $n-1$ `eval` steps starting from the `main` method such that:

1) $nextObject(h_i, s_i) = o_i$,
2) $\texttt{eval o}_i\ \texttt{h}_i = (\texttt{res}_i,\ \texttt{l}_i,\ \texttt{h}_{i+1})$,
3) $s_{i+1} = updL(s_i, o_i, l_i)$.

Then $\mathcal{C}(\mathcal{T}_E, o_i, \mathcal{M}) \leq UB^{o_i}_{\mathcal{M}}$ for every object o_i.

Proof:
By Theorem 4.3 there is a trace executing the same statements (recall that we assimilate S_i and res_i)

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

By Lemma 4.7 there is a trace $\mathcal{T}_C = \|{}^c[\![(h_1, s_1)]\!]^{-1}\| \leadsto^* \|{}^c[\![(h_n, s_n)]\!]^{-1}\|$ executing the same statements in the same objects as $\mathcal{T}$, therefore $\mathcal{C}(\mathcal{M}, o_i, \mathcal{T}) = \mathcal{C}(\mathcal{M}, o_i, \mathcal{T}_C)$ for every object o_i. Theorem 3 from [25] states that $\mathcal{C}(\mathcal{M}, o, \mathcal{T}_C) \leq UB^o_{\mathcal{M}}$ for every object o_i, and $\mathcal{T}_E$ executes the same statements as $\mathcal{T}$ in the same objects, so $\mathcal{C}(\mathcal{M}, o_i, \mathcal{T}_E) = \mathcal{C}(\mathcal{M}, o_i, \mathcal{T})$. Therefore, for every object o_i we can conclude that $\mathcal{C}(\mathcal{M}, o_i, \mathcal{T}_E) = \mathcal{C}(\mathcal{M}, o_i, \mathcal{T}) = \mathcal{C}(\mathcal{M}, o_i, \mathcal{T}_C) \leq UB^o_{\mathcal{M}}$. □

5. Experimental evaluation

In the previous section, we proved that the execution of compiled Haskell programs has the same resource consumption as the original ABS traces wrt. 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 left for 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 the source language's statements executed (steps) and

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

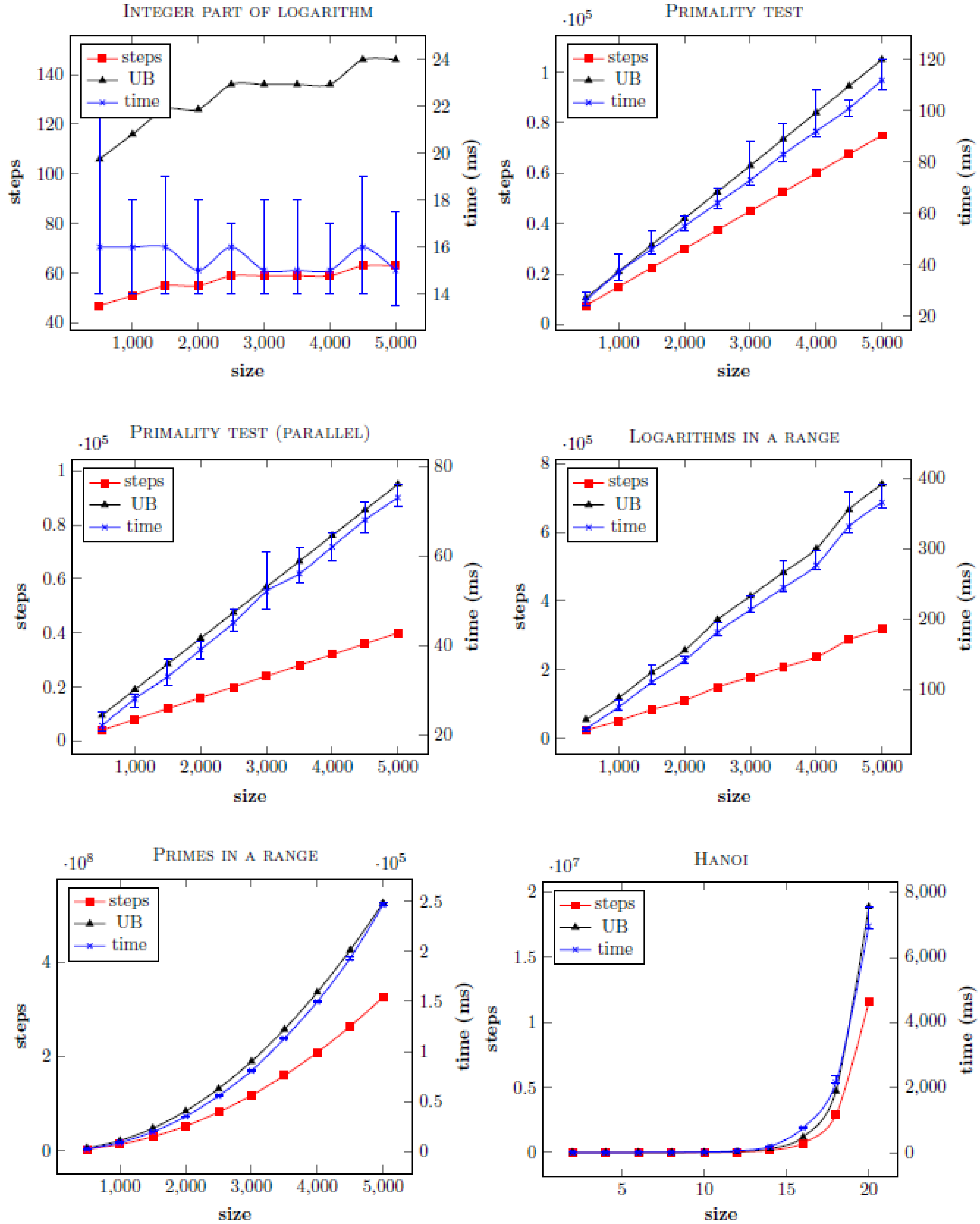

Figure 13: Execution steps, upper bound, and time (Intel® Core™ i7-4790 at 3.60GHz, 16 GB).

their wallclock runtime to program completion. The *Integer part of logarithm* is a simple approximation of the computation $log_2\ n$ with has the same complexity $O(log\ n)$. 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 parallelism* 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 *Logarithms in a range* program computes the integer part of the logarithm of n numbers, so it has cost $O(n.log\ n)$. *Primes in a range* computes the prime numbers in the interval $[1..n]$, thus having a $O(n^2)$ cost. Finally, the *Hanoi* program solves the Tower of Hanoi puzzle with n being the number of disks to be placed; the complexity of the solution to Hanoi is of the class $O(2^n)$.

We have tested the programs with n ranging from 500 to 5000 (except *Hanoi*, where due to its high complexity we have used sizes from 2 to 20), running 20 experiments for every value of n, and measured the time. This is plotted in the cross line (right margin) in Figure 13. 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 Figure 13.[5] 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. It is worth to note that the results to the *Integer part of logarithm* show a high margin of error in the measurements of the Haskell execution runtime, and thus cannot reveal the expected logarithmic grow of execution time. This happens for a number of reasons: to keep the size n uniform among the programs tested, the range 500 to 5000 is computationally very similar for the $log(n)$ problem. Second of all, our runtime measurement tool has a precision of a millisecond which is not enough for this particular range. Furthermore, we have measured the overall wall-clock time of executing the tested programs and that includes any garbage collection overhead incurred by the Haskell runtime system; the Haskell-GHC garbage collector is difficult to control and for that, we used a large allocation area so as to minimize its effects.

We have also plotted the resource bounds obtained by the SACO tool [9] for the different values of n (triangle line, left margin in Figure 13). SACO can analyze full ABS programs and thus also the subset considered in this article, 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, and they deviate from the actual cost by a multiplicative constant factor (i.e., the complexity class is preserved). Such a multiplicative constant deviation 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 and loops 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

[5] Note that the number of steps in the left margin is usually multiplied by some factor (10^5, 10^7, and 10^8) that appears on top of the margin.

cost (n statements) and generates an upper bound of n^2 instead of the more precise (but asymptotically equivalent) expression $1 + 2 + \ldots + n$.

6. Related work

The described target language is an untyped extract of the canonical ABS-Haskell backend [11], 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 [26] for concurrent Prolog. These engines encapsulate multi-threading by providing entities that evaluate goals and yield answers when requested. They follow a similar coroutining 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.

We used the Haskell language to implement the structural operational semantics of ABS for the execution of ABS programs. Different high-level executable languages are used to implement the operational semantics of programming languages, e.g., Maude [27] and K [28] which are based on rewrite logic, and the interactive theorem provers HOL [29] and Cog [30]. The Coq proof assistant has been used in [31] to prove that a compiler translating a subset of C programs to assembly code preserves the semantics. In this article, we have showed that the results of the static analysis of the resource consumption of ABS programs extended with a cost model also hold at runtime as provided by the Haskell language.

7. Conclusion and future work

We have presented an actor-based language with cooperative scheduling (a subset of ABS) and its compilation to Haskell using continuations. The compilation is formalized in order to establish that the program behavior and the resource consumption are preserved by the translation. Compared to the only other formalized ABS backend [2] (in Maude), our Haskell translation admits the preservation of resource consumption, and as a side benefit, makes uses of an overall faster backend.[6]

In the future, we plan to extend our formalization to accommodate full ABS, both in terms of the omitted parts of the language as well as the non-deterministic behavior of a multi-threaded scheduler,

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

e.g. by broadening our simulated scheduler to non-determinism, and perhaps (M:N) thread parallelism. Another consideration is to relate our resource-preservation result to a distributed-object extension of ABS [11]; 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.

References

[1] Lorincz K, Chen Br, Waterman J, Werner-Allen G, Welsh M. Resource Aware Programming in the Pixie OS. In: Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems, SenSys '08. ISBN 978-1-59593-990-6, 2008 pp. 211–224.

[2] Johnsen EB, Hähnle R, Schäfer J, Schlatte R, Steffen M. ABS: A Core Language for Abstract Behavioral Specification. In: FMCO '10, LNCS 6957. Springer, 2010 pp. 142–164.

[3] Flanagan C, Felleisen M. The Semantics of Future and its Use in Program Optimization. In: Proc. POPL '95. ACM. ISBN 0-89791-692-1, 1995 pp. 209–220. doi:10.1145/199448.199484.

[4] de Boer FS, Clarke D, Johnsen EB. A Complete Guide to the Future. In: Proc. ESOP '07, LNCS 4421, pp. 316–330. Springer. ISBN 978-3-540-71314-2, 2007. doi:10.1007/978-3-540-71316-6_22.

[5] Knuth DE. The Art of Computer Programming, Volume 1: Fundamental Algorithms, 2nd Edition. Addison-Wesley Professional, 1973.

[6] Albert E, de Boer FS, Hähnle R, Johnsen EB, Schlatte R, Tapia Tarifa SL, Wong PYH. Formal modeling and analysis of resource management for cloud architectures: an industrial case study using Real-Time ABS. *Service Oriented Computing and Applications*, 2014. **8**(4):323–339.

[7] Albert E, Arenas P, Fernández JC, Genaim S, Gómez-Zamalloa M, Puebla G, Román-Díez G. Object-sensitive cost analysis for concurrent objects. *Softw. Test., Verif. Reliab.*, 2015. **25**(3):218–271.

[8] Wong PY, Albert E, Muschevici R, Proena J, Schfer J, Schlatte R. The ABS Tool Suite: Modelling, Executing and Analysing Distributed Adaptable Object-Oriented Systems. *STTT*, 2012. **14**(5):567–588.

[9] Albert E, Arenas P, Flores-Montoya A, Genaim S, Gómez-Zamalloa M, Martin-Martin E, Puebla G, Román-Díez G. SACO: Static Analyzer for Concurrent Objects. In: Proc. TACAS '14, LNCS 8413, pp. 562–567. Springer. ISBN 978-3-642-54861-1, 2014. doi:10.1007/978-3-642-54862-8_46.

[10] Srinivasan S, Mycroft A. Kilim: Isolation-Typed Actors for Java. In: Proc. ECOOP '08, LNCS 5142, pp. 104–128. Springer, 2008.

[11] Bezirgiannis N, de Boer FS. ABS: A High-Level Modeling Language for Cloud-Aware Programming. In: Proc. SOFSEM'16, LNCS 9587, pp. 433–444. Springer, 2016.

[12] Palacios A, Vidal G. Towards Modelling Actor-Based Concurrency in Term Rewriting. In: Proc. WPTE '15, volume 46 of *OASICS*. Dagstuhl Pub., 2015 pp. 19–29. doi:10.4230/OASIcs.WPTE.2015.19.

[13] Vidal G. Towards Erlang Verification by Term Rewriting. In: Proc. LOPSTR '13, LNCS 8901. Springer, 2013 pp. 109–126. doi:10.1007/978-3-319-14125-1_7.

[14] Noll T. A Rewriting Logic Implementation of Erlang. *ENTCS*, 2001. **44**(2):206–224. Proc. LDTA '01.

[15] Albert E, Arenas P, Gómez-Zamalloa M. Symbolic Execution of Concurrent Objects in CLP. In: Proc. PADL '12, LNCS 7149. Springer, 2012 pp. 123–137. doi:10.1007/978-3-642-27694-1_10.

[16] Nakata K, Saar A. Compiling Cooperative Task Management to Continuations. In: Proc. FSEN'13, LNCS 8161, pp. 95–110. Springer. ISBN 978-3-642-40212-8, 2013. doi:10.1007/978-3-642-40213-5_7.

[17] Albert E, Bezirgiannis N, de Boer F, Martin-Martin E. A Formal, Resource Consumption-Preserving Translation of Actors to Haskell. In: Hermenegildo MV, Lopez-Garcia P (eds.), Proc. LOPSTR'16 (Revised Selected Papers), volume 10184 of *LNCS 10184*. Springer International Publishing. ISBN 978-3-319-63139-4, 2017 pp. 21–37. doi:10.1007/978-3-319-63139-4_2.

[18] Schäfer J, Poetzsch-Heffter A. JCoBox: Generalizing Active Objects to Concurrent Components. In: ECOOP '10, LNCS 6183, pp. 275–299. Springer, 2010.

[19] Dean J, Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 2008. **51**(1):107–113. doi:10.1145/1327452.1327492.

[20] Appel AW, Jim T. Continuation-passing, Closure-passing Style. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '89. ACM, New York, NY, USA. ISBN 0-89791-294-2, 1989 pp. 293–302. doi:10.1145/75277.75303. URL `http://doi.acm.org/10.1145/75277.75303`.

[21] Danvy O, Filinski A. Representing Control: a Study of the CPS Transformation. *Mathematical Structures in Computer Science*, 1992. **2**(4):361391. doi:10.1017/S0960129500001535.

[22] Nipkow T, Wenzel M, Paulson LC. Isabelle/HOL: A Proof Assistant for Higher-order Logic. Springer-Verlag, 2002. ISBN 3-540-43376-7.

[23] Paulin-Mohring C. Introduction to the Coq Proof-Assistant for Practical Software Verification, pp. 45–95. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-642-35746-6, 2012. doi:10.1007/978-3-642-35746-6_3. URL `https://doi.org/10.1007/978-3-642-35746-6_3`.

[24] Claessen K, Hughes J. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In: Proc. ICFP '00. ACM. ISBN 1-58113-202-6, 2000 pp. 268–279. doi:10.1145/351240.351266.

[25] Albert E, Arenas P, Correas J, Genaim S, Gómez-Zamalloa M, Puebla G, Román-Díez G. Object-Sensitive Cost Analysis for Concurrent Objects. *Software Testing, Verification and Reliability*, 2015. **25**(3):218–271. doi:10.1002/stvr.1569.

[26] Tarau P. Coordination and Concurrency in Multi-engine Prolog. In: Proc. COORDINATION '11, LNCS 6721. Springer. ISBN 978-3-642-21463-9, 2011 pp. 157–171.

[27] Meseguer J. Twenty years of rewriting logic. *J. Log. Algebr. Program.*, 2012. **81**(7-8):721–781.

[28] Rosu G. $\mathbb{K}$: A Semantic Framework for Programming Languages and Formal Analysis Tools. In: Dependable Software Systems Engineering, pp. 186–206. IOS Press, 2017. doi:10.3233/978-1-61499-810-5-186.

[29] Nipkow T, Paulson LC, Wenzel M. Isabelle/HOL - A Proof Assistant for Higher-Order Logic, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[30] Bertot Y, Castéran P. Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[31] Leroy X. Formal verification of a realistic compiler. *Commun. ACM*, 2009. **52**(7):107–115.