HLogo: A Haskell STM-Based Parallel Variant of NetLogo

Nikolaos Bezirgiannis[1](✉), I.S.W.B. Prasetya[2], and Ilias Sakellariou[3]

[1] Centrum Wiskunde & Informatica (CWI), P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
`n.bezirgiannis@cwi.nl`

[2] Department of Information and Computing Sciences, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
`s.w.b.prasetya@uu.nl`

[3] Department of Applied Informatics, University of Macedonia, 156 Egnatia Street, 54636 Thessaloniki, Greece
`iliass@uom.edu.gr`

Abstract. Agent-based Modeling and Simulation (ABMS) has become a quite popular approach among researchers in the community, mainly due to its simplicity, expressiveness and wide applicability. However, in most cases, ABMS tools demonstrate reduced performance, especially when dealing with large experiments. This paper presents HLogo, a parallel variant of the NetLogo ABMS framework, that aims to increase the performance of simulations by utilizing Software Transactional Memory and multi-core CPUs, while maintaining the user friendliness of NetLogo. HLogo is implemented as a Domain Specific Language embedded in the functional language Haskell, which means that it also inherits Haskell's features, such as strong static typing, a module system and a vast collection of programming libraries.

Keywords: Agent-based Modeling · Agent-based simulation · Concurrent agent-based simulation · Concurrent NetLogo.

1 Introduction

Simulating a system as a set of interacting agents has gained significant popularity in the past decades. The approach proved to be both natural and widely applicable in various areas, giving rise to the so called Agent Based Modelling and Simulation (ABMS) field. In the latter, complex system behaviour often emerges from the interaction of relatively simple agents and simulation allows to study these emergent phenomena. ABMS has been shown to have broad applicability, e.g. in social sciences [13], ecology [15], biology [28], and physics [39]. As a consequence, numerous ABMS frameworks have been proposed [6,29,36], and research has focused on various aspects of the latter, such as methodology [32], ease of use [40], portability [2], and expressiveness [31]. The size of agent populations may be a crucial factor towards the emergence of certain sought-after system phenomena, so performance is also an important aspect. The ability to scale up simulations to large populations comes down to, basically, how much computation the used ABMS framework can "crunch" per time unit. However, till recently little attention was devoted in improving the performance of available ABMS frameworks [30].

This paper extends our previous work reported in [4] by introducing in greater depth a new ABMS framework called HLogo. The framework is strongly inspired by the well-known NetLogo framework [38]. Like NetLogo, HLogo also strives for simplicity. For this reason HLogo is implemented, actually embedded, in a powerful and pure functional programming language called Haskell [1], hence the name "HLogo". Unlike NetLogo, HLogo is a *concurrent* ABMS framework, that aims to speed up simulations by harvesting the parallelism available in modern multi-core CPUs. HLogo offers three unique features which also constitute the main contribution of this paper:

- HLogo allows agents to run *concurrently*, with the latter implemented by utilizing a technology called Software Transactional Memory (STM) [33]. This allows the complexity of synchronizing agents to be completely hidden from the programmers, hence keeping HLogo just as simple as NetLogo, despite the concurrency. Coupled with Haskell's lightweight (green) threads, the overall ABMS execution enjoys significant benefits from *multi-core* parallelism.
- HLogo is *embedded* as a Domain Specific Language (eDSL). As a DSL it has a simple base syntax. As an embedded DSL it also inherits all the advantages of its host language, Haskell. For example, it inherits Haskell's module system (which NetLogo lacks), allowing HLogo programmers to import and use a plethora of Haskell libraries. The decision to embed the DSL, rather than directly implementing it, does mean however that its syntax is limited by that of its host language.
- HLogo is statically typed and it inherits Haskell's *type inference*. This strengthens the safety of HLogo programs without burdening the user with writing type annotations.

The rest of this paper is organized as follows. Section 2 outlines the NetLogo approach to ABMS, in order to explain the concepts implemented in HLogo. Section 3 presents related work in the area of Logo-like simulation platforms and parallel approaches to simulation. Some important features of Haskell, the implementation language of choice, are presented in Sect. 4. The HLogo language is presented in Sect. 5 with Sect. 6 discussing the parallel features of the former, and Sect. 7 reporting an experimental evaluation of the former. Finally, Sect. 8 concludes the paper and presents future research directions.

2 Agent Based Modelling and Simulation in NetLogo

The ABMS community has enjoyed the introduction of a significant number of simulation platforms that differ in a number of characteristics, such as modelling approach, efficiency, user-friendliness, etc. NetLogo [38] is one of the most well known and widely used platforms, mainly due to its simplicity, small learning curve and the ease by which users create *simulations*, or *experiments* as the former are referred to in NetLogo.

Three kinds of agents exist in the NetLogo environment, namely *patches*, *turtles* and *links*, each modelling a different aspect of the simulation world. All entities are stateful and active, i.e. they have a set of attributes (variables) which determine their state and the user can encode agent behaviour using the provided domain specific language. The latter offers a set of primitives that target encoding of both agent perception and action mechanisms, simplifying the task significantly.

Patches are stationary agents, that form the two dimensional (or three dimensional) world, i.e. the grid on which turtles "live". The dimensions of the grid are fixed during the execution of an experiment and in the case of patches, variables and code allow modelling of complex environments. Turtles are agents that are dynamically created during the experiment and can move on the grid with links connecting two turtles, i.e. representing a relation between the latter. Agents can be organized into breeds, although the depth of such organisation is limited to one. Each breed can have its own user-defined attributes, apart from the system-specified ones. For instance, this declaration:

```
breed [cows cow]
cows-own [age hunger]
```

defines a new breed called `cows`, with each of its member having attributes `age` and `hunger`.

One of the important notions in programming NetLogo agents is that of an *agentset*, i.e. a set of agents of a specific type (either turtles, patches or links) that have certain characteristics. For instance, all agents of a specific breed are part of the agentset with the same name (e.g. `cows`). More interesting agentsets are formed with the use of NetLogo primitives, as for example `cows in-radius 3` which forms an agentset of all cows located around the calling agent at a specific distance. A rich set of primitives similar to the above allow implementation of agent perception mechanisms. Among those are the `with` primitive that defines a boolean condition on the agentset formation, i.e. the line `cows with [age>4]`, will collect all cows older than 4 years old.

Execution involves *asking* an agent or a set of agents to perform some action. For instance, the following line of code:

```
ask cows in-radius 3 [set hunger 0]
```

commands all cows inside the specific radius to set their hunger attribute to 0. The *observer* entity is the initiator of the experiment, and can ask other agents to execute some code, and every agent can ask other agents to perform some action. The built-in variables `self` and `myself` refer to the currently executing agent (similar to 'this' in OO), and the parent caller that `ask`ed this agent, respectively. Besides built-in commands an agent can execute custom commands defined by user-defined *procedures* and user-defined functions called *reporters*, in NetLogo terminology.

Finally, the state of other turtles can be queried by using the "of" primitive. For example, the expression [age] of cows reports back a list of all cows' ages in the simulation world. In addition to reporting an attribute, "of" can also evaluate a function on the target agent's context and report the result.

The approach, briefly outlined above, has become widely accepted by researchers using ABMS in a number of fields, with the list of publications citing NetLogo becoming quite large and increasing steadily each year, proving that the environment has sufficient modelling capabilities. However, building large experiments is currently not sufficiently supported, mainly due to the fact that the execution model is sequential. During an ask command as the one shown above, each agent in the set has to complete the execution of the code given to it, before the next agent in the set can start. Although this approach does solve a number of problems, it simply cannot take advantage of the computational power offered by current multi-core processors.

Running a simulation in parallel is a rather complex task, since executing agents have constant access to a shared environment as well as each other's states. Handling concurrent changes creates a major challenge that any parallel simulation platform must address. This work addresses the problem by introducing a NetLogo variant implemented in Haskell, relying to the Software Transactional Memory control mechanism provided by the later.

3 Related Work

Since this work deals with a parallel ABMS platform developed in Haskell, this section first discusses existing implementations of NetLogo ABMS platforms, then parallel simulation platforms, and finally existing simulation frameworks in Haskell.

NetLogo [38] models are written in a dialect of the educational programming language Logo. The language is dynamically typed with lexical variable scoping and is implemented in the Scala programming language. A compiler translates NetLogo code to Java bytecode, to be later run in a Java Virtual Machine (JVM). The platform includes a GUI to visualize simulation results, and a rich collection of predefined models. A limited form of type checking based on agent types (turtles, patches etc.) is supported, i.e. there are certain commands that can only be run in a specific agent context and a user defined procedure that contains them must be run in the same context as well.

ReLogo [24] is a NetLogo clone embedded as a DSL in Groovy (an OO language running in JVM) that comes as a part of the Repast simulation suite. As is the case with NetLogo, ReLogo is single-threaded and comes with a rich GUI & IDE based on Eclipse. Although Groovy 2.0 introduced optional static typing, ReLogo cannot type-check many of its expressions: agentsets are untyped, and ask/of/with closures cannot track the type information of their context (self, myself). The simulation user has to either resort to type-casting or turn off Groovy's static typing in the pertinent code.

In order to speedup large sets of experiments, both NetLogo and ReLogo support *parameter sweeping* [19]: running *multiple instances* of the same model in parallel, each on its own CPU core, while varying the model's input parameters. Taking a more traditional approach, HLogo, tries to inject parallelism *inside a single instance* of a simulation run; this can be crucial for large models or time-critical simulations where any performance gain is desirable.

To the best of our knowledge, the work described in this paper is the first to apply Software Transactional Memory in Agent-Based Modelling. However, there are other approaches in the literature that investigate the issue of speeding up ABM execution using various parallel techniques: the work described in [21] proposes to execute Agent-based systems through Distributed Discrete-Event Simulation. The key problem as reported in the paper, is the decomposition of the environment which leads to the problem of fair load balancing of the distributed machines. SPADES [30] is another Distributed Agent Simulation Environment that explicitly models the full agent cycle (sense-think-act), while having distributed execution and reproducibility of results. The work reported in [22] employs a well known parallelization technology, OpenMP, to speed up the execution of agent-based models. However, the technique restricts the implementation language of ABM frameworks to only those which provide an OpenMP implementation, i.e. C, C++, Fortran. It also adds the burden of annotating simulation code with extra OpenMP pragmas, which is rather discouraging for simulation developers. SASSY [16] is a scalable agent based simulation system that acts as a middle-ware between an agent-based API and a Parallel Discrete Event simulation (PDES) kernel. The difference in SASSY compared to [21,30] is that the ABM framework can be built up from existing standard PDES kernels. An innovative method of executing mega-scale Agent-Based Models in the massively parallel Graphics Processing Unit (GPU) is proposed in [10]. Although, it is well established that this method can lead up to considerable speed gains, we feel that the expressiveness of Agent-based models that can be run on this platform is rather restricted. A similar framework is Flame GPU, built on-top the FLAME ABM framework [17], and has successfully been applied on project EURACE to simulate the European economy model [8].

Within the Haskell community, our work is the first Logo-based simulation framework implemented in Haskell. Other simulation packages for Haskell are for example *Hasim* [3] and the recently introduced *Aivika* [35]; both are libraries for Discrete Event Simulation (DES). *Hasim* [3] provides process-based DES, however it does not employ any kind of parallelism. *Aivika* provides DES with extensive system dynamics. There is also the *event-monad* library that provides a monad (see also Sect. 4.2) and monad transformer for events; it can be used as a low-level helper library to build a simulation framework. Users can create an event-graph simulation system and schedule events to it. In principle, it does not employ any parallelism, but it could theoretically be used together with some parallel strategy to exploit parallelism.

4 Haskell

The functional programming language Haskell [1] was selected as the HLogo implementation platform, for two main reasons. Firstly, it is an excellent choice for embedding domain specific languages (DSLs) [14] and HLogo is designed as a DSL. Secondly, Haskell offers an excellent implementation of Software Transactional Memory (STM) [9], which is crucial for realizing HLogo's parallelism. This section will introduce several concepts from functional programming necessary to explain the embedding of HLogo in Haskell, and introduce STM.

4.1 Static Typing

An important feature that HLogo gets from its implementation in Haskell is that the latter is a statically typed language. This extends to HLogo as well, which is in contrast to NetLogo's dynamic typing. For example, in Haskell, and thus also in HLogo, a type error in an expression such as `1 + non_number` will be detected at compile time. It should be noted that NetLogo also checks type consistency of its expressions, but the majority of such checks is done at runtime and consequently, such errors, as in the example above, are detected rather late. In this respect, HLogo can be said to provide more safety for agent-based modeling. However, if the user needs dynamic typing, e.g. for its flexibility, it can be supported in Haskell through the Data.Dynamic module.

HLogo also takes advantage of Haskell's powerful type system and thus can type-check not just simple arithmetic expressions, but also more elaborate statements. The majority of built-in Logo-like commands need to be executed in a specific agent context, e.g. the command `forward` n may only be executed by a turtle agent (other types of agents are immovable). The example below presents a NetLogo expression, and its HLogo counterpart, where we erroneously "ask" the patch at location (0, 0) to die:

```
%% NetLogo version: yields error only later at runtime
ask patch 0 0 [die]

-- HLogo version: this program does not type-check
ask (atomic die) =<< patch 0 0
```

In this case, HLogo will detect the error successfully at compile time. Elaborating more on this error, the action `die` should only be invoked on either a turtle or a link, whereas patches are to live through out the simulation, so they should not "die". In Haskell, this is enforced by overloading the name `die`, which is achieved by defining `die` as an operation of a 'type-class' called `TurtleLink`. Haskell *type-classes* offer a similar concept to interfaces in OO languages, e.g. Java, used for ad-hoc polymorphism. A type-class defines a set of operations, but it only defines their signatures, and thus provides no implementation. When the type T is declared to be an instance a type-class, e.g. `TurtleLink` (in OO jargon we would say T 'implements' `TurtleLink`), T gets all `TurtleLink`'s operations including `die`, but on the other hand the declaration must specify how

T implements those operations. In HLogo turtles and links are declared to be instances of `TurtleLink`, thus allowing `die` to be overloaded for both types of agents. On the other hand, patches are not instances of the former type-class; therefore the action `die` in the above code yields a type error.

Haskell's strong type system also comes with Hindley-Milner type inference [23], which makes type annotations optional. This provides type safety to the user, without the burden of annotating the code with type signatures. For example, the following command is a variation of the previous example:

```
ask (atomic (do {forward 3 ; die})) =<< agentT
```

There is no need to explicitly annotate the type of `agentT`. Haskell type system can infer that it must be a turtle: it "knows" that `die` expects an agent which is an instance of `TurtleLink`, whereas `forward` expects a turtle (patches and links are not supposed to move around). So by implication Haskell successfully infers that the agent on which the above commands are applied has to be of type Turtle.

4.2 Monads

In Haskell, types can be parameterized. For example, lists of integers are of type [`Int`]. Actions that perform input/output (IO) are instances of the type `IO` a, where a is the type of the result of such an action. For instance, the function `getChar` that reads from the console and returns the read character has the type `IO Char`. The [.] and `IO` parts in these examples are called *type constructors*; they construct a new type from the type given to them.

Being a purely functional language, Haskell has no natural concept of side effect and thus modeling agents that are stateful and have actions with side effects on their own or other's states and the environment does not come naturally. However, such states and side effects are brought into the language through monads [27]. In functional programming, a *monad* is a generic concept for composing items through an associative operator. A simple example of a monad is lists with the concatenation operator. It should be noted that in Haskell function compositions are allowed and thus monads can be used to compose computations. In more technical terms, a Haskell `Monad` is a type-class and for the purpose of this discussion, we will assume that this type-class offers an associative operator ";" for composing monadic actions[1]. A type constructor `M` can be made an instance of `Monad` by providing a concrete definition for ";". If `M` is a monad, then an expression of type `M` a is a monadic action: when executed, at the end will produce a value of type a. For example, the previously mentioned type constructors, `IO` and [.] are both monads and the function `getChar` is thus a monadic action. The set of commands for HLogo agents, such as `forward` and `die` also form monads.

[1] The actual definition of `Monad` offers a slightly different and richer set of operations [26].

Monadic actions can be *sequentially composed* with the **do** notation. E.g., suppose c is a monadic action of type `M Int`, then the expression: **do**{`z` ← `c`; **return** (`z+1`) } is a new monadic action[2]. It first evaluates/executes c, then binds the produced value in the variable z, then $z+1$ is returned as the produced value of the whole action. The concept is general through the overloading of ";" operator (recall that `Monad` is a type-class) and the exact meaning of the example **do** expression above depends on the used monad (which `M` is being used). If it was the IO monad, or the monad of agents' commands, the meaning is roughly as described above. If the monad was the list monad, it would have the net effect of constructing the list $[\, z+1 \mid z \in c\,]$.

The expression **do**$\{e_1\ ;\ e_2\ ;\ ...\}$ will evaluate the expressions e_1, e_2, ... in the given order. When e_1, e_2, ... are written vertically, and start at the same column, we can drop the use of delimiters "{","}", and the ";" to get a cleaner syntax. A **do**-sequence that does not explicitly specify a **return** as the last expression, implicitly returns whatever the last expression returns. For example, the HLogo expression below is a monadic action that first obtain the set of all patches in the simulation, and then it returns the number of elements of this set.

```
do p ← patches
   count p
```

There is also the e =<< d operator that we saw before in Sect. 4.1, that pipes the value returned by the monadic action d as an input for e. So, the above code can also be written more succinctly as `count =<< patches`.

4.3 Software Transactional Memory (STM)

STM is a concurrency control mechanism that allows concurrent processes to be programmed without having to synchronize them, and yet allowing them to safely operate on common states. This makes the task of programming such processes much easier and much less error prone. There is no need to worry about race conditions, nor deadlocks. STM's concurrency relies on the so-called *transactions*—a concept that originally comes from the database domain. A transaction is a sequence of reads and writes to a set of so-called *transactional variables* or *TVars*. A TVar represents an actual store in the memory, which can be shared by multiple transactions. The execution of a transaction is *virtually atomic*, that is, its intermediate changes on the TVars it operates on cannot be witnessed by any other transaction. The idea originates from the field of Distributed Databases, where a transaction corresponds to an atomic SQL query or update. Whereas database transactions operates on tables' rows or columns, transactional memory operates at the level of memory stores. Historically, transactional memory was introduced in 80's by Knight as an extension to Lisp with suitable hardware modifications to enable concurrency [18]. In 90's, Shavit and Touitou turned the idea to a pure software implementation of the approach, and coined the

[2] The actual symbol in Haskell is <- instead of ←. We use the later just for improving the presentation.

term Software Transactional Memory. Recently, STM programs can be further accelerated through hardware instruction-set extensions, e.g. with Transactional Synchronization Extensions (TSX) of the Intel® Skylake processor.

Multiple transactions can run in parallel, however each transaction τ does not directly write to its TVars. Instead, it keeps a separate log of reads and writes to the TVars, with writes not committed yet. At the end of the transaction, it checks if one of its TVars has been modified with respect to its value at the start of τ. If no modifications have occurred, all τ's writes are committed and the transaction is said to be successful. Otherwise, τ is aborted, and later retried again.

Haskell has an excellent STM library [9] and furthermore, its strong type system guarantees that transactions are ‘safe’. It is important that aborted transactions *have no irreversible side effects*, in other words, if they do have effects, we should be able to rollback those effects. This can present a problem, as for example in cases where within a transaction we perform IO actions (e.g. to read a value from the keyboard), which typically cannot be rollbacked. In most other language implementations, this cannot be enforced. In Haskell however, STM transactions and IO actions are both monadic actions, but of different types: a transaction that returns an integer will have the type `STM Int` whereas an IO action that read an integer from the keyboard has the type `IO Int`. Haskell type system guarantees that expressions of these monads cannot be intermixed, in the same way that Haskell does not allow an instance of `Int` to be subtracted from an instance of `Bool`.

In HLogo, a model typically consists of many agents. To gain parallelism, agents' actions can be composed from transactions. Internally, committing a transaction involves complicated orchestration where different places in the memory must be locked and unlocked in a certain order. However, this process is hidden from the programmers: from their perspective transactions are lock free, thus making concurrent programming much easier for them. For HLogo this is important, since we want to maintain NetLogo's user friendliness. Using STM does have its price. It introduces overhead due to aborted transactions. But still, experiences reported with STM in the literature suggests that considerable speedup can still be expected [25].

Ultimately, transactions are run by threads for concurrent execution. There are several options on how to do this. The obvious one is to run each transaction on its own thread. Haskell provides so-called green threads, i.e. virtual threads managed by a virtual machine (or by a language's runtime-system), as opposed to OS' native threads. Green threads use less memory and can be activated and synchronized faster. A large number (thousands; even millions) of green threads can be created without running out of memory. Haskell runtime-system employs an M:N threading model, where M green threads are automatically mapped to N OS (heavy weight) threads for multi-core parallelism. While this maximizes concurrency, in our case most of the time the number of available CPU cores is much less than the number of agents. The above solution would lead to performance degradation. Section 6 will discuss our solution to this.

5 HLogo

Both NetLogo and HLogo are instances of Domain Specific Languages (DSLs) for describing and simulating dynamic systems. A DSL is a programming language that offers, through appropriate notations and abstractions, expressive power focused on a particular problem domain [37]. NetLogo is a *native* DSL, i.e. it has a dedicated parser and a compiler or interpreter to execute its code, whereas HLogo is an *embedded* DSL (eDSL). An eDSL is embedded inside another language (host), usually a general purpose programming language and tries to *imitate* a native DSL by providing its language constructs in terms of the constructs of the host. It is an imitation in the sense that it may not look or even work entirely the same as the DSL it imitates, but it tries to. On the other hand, an eDSL can be more rapidly developed because we do not need a separate parser and compiler for it. An eDSL also inherits all the important features of its host language. HLogo inherits, among other features, Haskell's:

– expressiveness and its powerful type checking, as discussed in Sect. 4.1,
– module system, thus allowing us to organize HLogo agents into separate modules, a feature that NetLogo currently (as of version 5.3.1) lacks, and
– the already vast collection of open-source Haskell libraries (Hackage).

Haskell was chosen since it is a brilliant host language for embedding DSL's [14], as has been demonstrated in various cases [5,11,26]. In particular, the expressiveness provided by higher-order functions and type classes is crucial for imitating native DSL constructs. This approach also makes HLogo more easily extendible: new constructs can be simply added by the inclusion of more higher order functions, whereas in a native DSL we would need to modify the parser and interpreter.

It is true though, that as an eDSL the syntax of HLogo will be limited by the syntax of its host language, Haskell. For example, in NetLogo binary operators have higher precedence than function application; e.g. we can write: `print 1+3`. This is not possible in HLogo, because in Haskell the precedence is reversed. So, the same code in HLogo has to be written as: `print (1+3)`. As mitigation, Haskell's clean syntax and support for overloading can often be exploited to provide acceptable syntactical imitations of the original NetLogo constructs.

Nearly the complete set of NetLogo's standard library has been ported to HLogo. In the sections that follow, we limit ourselves to explaining how the main concepts are represented in HLogo. A complete example of a simple model is also included.

5.1 HLogo Agents

By default in each simulation there are always turtles, patches, and links. Identifiers of the corresponding names can be used to refer to them, e.g. `turtles` represents the set of all turtles in the model at hand. The dimension of the simulation world is set through command line and this determines the number of patches in the model. Turtles can be created dynamically e.g. by the command `create_turtles` N. The command `ask create_link_with` $\beta \mathrel{=\!\!<\!\!<} \alpha$ creates a link between α and β. These agents come with a number of pre-defined properties. E.g. turtles and patches have x and y coordinates specifying their position.

We can introduce a new breed of turtles and define new attributes for them. Technically, this requires the user to define a new Haskell datatype representing the breed, along with the corresponding set and get functions to access its attributes. To avoid having to write such boilerplate code, we employ Template Haskell [34], a compile-time meta-programming technique that will generate the needed code. As an example, the following code is part of the preamble of the example HLogo model in Fig. 1:

```
-- HLogo eDSL is a library
import Language.Logo

-- generates: cows,cows_here,...
breeds ["cows", "cow"]

-- generates getter/setter: energy
breeds_own "cows" ["energy"]
```

`breeds` and `breeds_own` are actually Template Haskell macros. The first creates, among other things, Haskell identifiers named `cows` and `cow` which can be used to refer to all cows, or to a specific cow (e.g. as in `cow 0 0`, the cow at position 0, 0). The second creates a new attribute for cows, which can be referred to by the identifier `energy`.

5.2 Commands

As mentioned in Sect. 2, NetLogo's main primitives for invoking commands on agents are: `ask`, `of`, and `with`. HLogo also provides these primitives (since "`of`" is a keyword in Haskell, the name "`of_`" is used instead).

The general syntax of `ask` is `ask` $c \mathrel{=\!\!<\!\!<} \alpha$, where c is the command to invoke and α is an agent (or an agentset) and which has the obvious effect of invoking c on α (or all members of α). More precisely, a *command* like c is a monadic action. The monad has a quite rich structure, but abstractly we can view commands as instances of the `IO` monad. For example `xcor` and `ycor` are commands that return respectively the x and y coordinates of the given agent. Other examples include `forward` and `die` mentioned in Sect. 4.1. These are actually STM transactions, which can be turned into commands by wrapping them with the function `atomic`—the connection will be discussed later. The whole construct `ask` $c \mathrel{=\!\!<\!\!<} \alpha$ is again a command; it simply returns void.

Since these are monadic actions, commands can be composed with the **do** notation (see also Sect. 4.2), e.g. **do** $\{c_1; ...; c_n\}$, and we can invoke them on a set of agents, e.g. all turtles, as in:

`ask` (**do** $\{c_1; ...; c_n\}$) $\mathrel{=\!\!<\!\!<}$ `turtles`

The primitives `of_` and `with` can be used with similar syntax: `of_` $c =\!\!\ll \alpha$ and `with` $c =\!\!\ll \alpha$. The primitive `of_` is actually just a slight variation of `ask`; whereas `ask` always returns void, `of_` $c =\!\!\ll \alpha$ returns whatever value c returns. If α is an agentset, the construct then returns a list of the results of invoking c on every member of α. The primitive `with` expects c to be of type `IO Bool` and α to be an agentset. It returns a new agentset consisting of those members of α, on which c returns true.

5.3 Procedures

NetLogo allows procedures to be defined through the `to` ... `end` syntax, for instance the code below defines the `move`[p] procedure, which will turn all cows 5° to the right, then move them p points forward:

```
to move[p]
   ask cows [right 5  forward p]
end
```

In HLogo, the same definition is achieved by a top-level function bound to its corresponding right-hand side monadic action:

```
move p  =  ask (atomic (do {right 5 ; forward p })) =<< cows
```

5.4 A HLogo Model Example

A complete model simulating a population of cows living on a field is shown in Fig. 1. In this simple model, Grass grows on random patches in the field and cows move around randomly, eating grass to gain energy. Regrowth of the grass and loss of energy are not included in this simple model. The example code also demonstrates a rudimentary support for visualization: the command `snapshot` can be called at any place in HLogo code to save an image of the current simulation's 2D canvas to a fresh postscript image. The image in Fig. 2 shows a snapshot of a run of the model in Fig. 1. Live visualization, as offered by the NetLogo platform, is part of future work.

6 Parallelizing HLogo

Both HLogo and NetLogo models are compiled to native code to run simulations. HLogo uses the Haskell compiler whereas NetLogo is actually compiled to Java bytecode which is then interpreted by a JVM; however, nowadays JVMs regularly employ Just-In-Time (JIT) compilation to native code. Both HLogo and NetLogo's simulation engines use similar data-structures to store agents: a 2-dimensional array for patches, and tree-based maps for turtles/links. However, they differ fundamentally on how they execute their commanding primitives (`ask`/`of`/`with`). E.g. in NetLogo's `ask` A c where A is an agentset, the command c is invoked on every agent in A, in a *sequential* manner. NetLogo does provide a variant called `ask−concurrent`; but this only simulates concurrency by

```
setup = do
    ask (do c ← one_of [green, brown]
            atomic (set_pcolor c)
         ) =<< patches
    cs ← create_cows 50
    ask (do x ← random_xcor
            y ← random_ycor
            atomic (do set_color white
                       set_energy 50
                       setxy x y
                   )) cs
    reset_ticks

go = forever (do
    t ← ticks
    when (t > 1000) stop
    ask (do {move ; eat_grass}) =<< cows
    snapshot
    tick)

move = do
    r ← random 50
    atomic (do {right r ; forward 1})

eat_grass = atomic (do
    c ← pcolor
    when (c==green) (do set_pcolor brown
                        e ← energy
                        set_energy (e + 30)))
```

Fig. 1. An example of agent model in HLogo. Cow-turtles move around and eat grass-patches to gain energy.

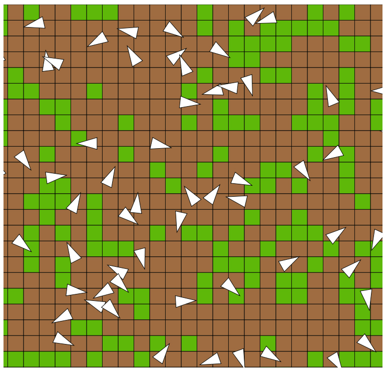

Fig. 2. An example of HLogo visualization output. White triangles represent cows. Green patches are patches with grass. Brown patches have no grass.

interleaving the execution of c between the agents in A. In other words, it does not run the agents in parallel. In contrast, HLogo tries to parallelize the execution of ask/of/with by utilizing Software Transactional Memory (STM) and green threads, two technologies for parallelism provided by Haskell discussed in Sect. 4.3.

Figure 3 shows the algorithm of the worker function behind HLogo's ask. The function $askWorker$ gets the command c to execute and an agentset A. It also gets a few other things as context: $self$ is the id of the agent that calls it and $parent$ is the id of $self$'s parent (in NetLogo also denoted by myself). The worker adopts a 'divide and conquer' strategy: it randomly splits A into N subsets, called *slices*, where N is the number of available CPU cores. For each slice B, a separate green thread t will be created, and the command c will be executed on every agent β in B. So, the execution of the slices is parallel, but within each slice the agents execute sequentially. Executing c on an agent β will need a context similar to the context passed to askWorker, except that β is now the 'self', and the worker's $self$ is its parent. Finally, the worker will wait until all slices finish their execution.

To randomly split A we use a high-quality treefish random generator library [7]. This random generator is splittable, allowing the random generator used by $askWorker$ to be split into N fresh generators, one for each slice so that the threads do not have to compete on a single generator. Note that c may contain another ask, which then needs a random generator to do its own splitting.

Notice that $askWorker$ blocks at the end, to wait until all the threads it spawned have completed. There is also a non-blocking variant of ask named ask_async where the worker simply continues.

```
$askWorker\ (self, parent)\ c\ A\ =$
    $N \leftarrow$ the number of CPU cores
    $slices \leftarrow$ use $rndG$ to split $A$ into $N$ parts
    $T \leftarrow \emptyset$
    **for** $B \in slices$
        $t \leftarrow (\lambda() \rightarrow$ **for** $\beta \in B \rightarrow$ executeCommand $(\beta, self)\ c)$
        run $t()$ as a new thread
        $T \leftarrow T \cup \{t\}$
        $i \leftarrow i+1$
    wait until all threads in $T$ finish
```

Fig. 3. The algorithm of HLogo 'ask' implementation.

The other two primitives, of_ and with, are implemented in a similar manner, with the only difference being that their workers need to collect (of_) and filter (with) the results of executing c on the agents in A. All three primitives are themselves commands, which implies that their usage can be nested. For example, in:

$$\texttt{ask (ask eat\_grass =<< cows\_here) =<< green\_patches} \qquad (1)$$

This command will ask every green patch to pass back all the cows that are currently on the patch, and ask these cows to eat the grass there. Note that this nesting will have maximum $N^2 + 1$ green threads running. Haskell runtime system will automatically load-balance the threads to the available CPU cores, if for example there are some patches with less or no cows on them.

6.1 Commands and Transactions

A Haskell thread expects to execute some code with IO effects. Consequently, commands are instances of the `IO` monad. However, if this simple approach is followed, race conditions might occur between the agents. Consider the following HLogo 'procedure' (in Haskell terminology this is a 'function') defining the command `eat_grass`:

```
eat_grass = do g ← grass
               when (g > 30) (do set_grass (g − 30)
                                 e ← energy
                                 set_energy (e + 30))
```

If we allow the commands in `eat_grass`'s body to operate in the IO monad, two race conditions may happen: (a) two cows eat grass from the same patch, but the patch grass level is decreased only once; (b) at another point in the program, an agent 'ask's to (destructively) modify the energy of a currently-eating cow.

Instead, what Hlogo actually does is to store agents' attributes in TVars, i.e. transactional variables as discussed in Sect. 4.3. Basic agent commands (e.g. `right`, `left`, `forward`, but also getters and setters such as `grass` and `set_grass` in the example above) are allowed to execute *only* inside an STM transaction. In other words, these commands are instances of the `STM` monad. As discussed in Sect. 4.2, using the **do**-notation we can compose multiple monadic actions to form a more complicated monadic action. This also applies to STM transactions. This means that the command `eat_grass` is an instance of `STM`. The code in (1) is thus not type correct since `ask` expects an instance of `IO` as the command.

We extend the language with the command atomic which given an STM transaction will try to 'run' it; when the atomic succeeds, it means its effects have been committed as a whole to the outside (IO) world, and will not be rollbacked. The type of atomic is `STM` $a \rightarrow$ `IO` a. So abstractly it is a function that turns an STM transaction into an instance of the `IO` monad. To fix (1) we can do the following, which is now type correct, parallel, and race-condition free (we underline 'atomic' to make it stand out):

$$\mathsf{ask}\ (\mathsf{ask}\ (\underline{\mathsf{atomic}}\ \mathtt{eat\_grass}) =\!\!\ll \mathtt{cows\_here}) =\!\!\ll \mathtt{green\_patches} \qquad (2)$$

Does this mean that the programmer should create as large transaction blocks as possible and merely surround them with a single `atomic`? Not exactly, since larger transactions can affect performance negatively, since the probability to conflict with transactions running on other threads increases. If a large transaction has to be rollbacked, the computation it has performed up to the point of rollback is also wasted. With HLogo, it is left to the programmer to decide if the whole transaction should be atomic or if it is safe to break it into smaller atomic blocks. As an example, below is a variation of `eat_grass` that avoids the above mentioned race-condition (a) and is faster than the variant with top level `atomic` in (2). It does not however avoid the race-condition (b).

```
eat_grass = do g ← atomic grass
               atomic (when (g > 30) (do set_grass (g − 30)
                                         e ← energy
                                         set_energy (e + 30)))
```

Despite the gain in parallelism, STM is not a ‘silver bullet’ to all problems that occur in a parallel setting. The execution of multiple STM transactions is inherently non-deterministic. In the simplest case, two simultaneous threads competing to modify the same TVar do not always commit in the same order. Consequently, HLogo simulations are non-reproducible, but still consistent with respect to race-conditions. On the bright side, HLogo’s engine guarantees that on a 1-core configuration, and if we do not use any asynchronous primitive such as the previously mentioned `ask_async`, the simulation of any agent model is reproducible.

7 Experiment

To compare the performance of HLogo to that of NetLogo, we ran the following benchmarks. They are run on 100×100 patches, forming a torus-shaped canvas. The benchmarks are simulated for 1000 ticks.

1. The benchmark *Redblue* has N turtles. The patches are randomly colored red or blue. At every tick, each turtle moves one step forward, and then turns $30°$ to the left if it is on a red patch, and else $30°$ to the right. Agents never write to the same TVar, and therefore their transactions never need to roll back.
2. The benchmark *Cows* has N cows. The patches are seeded with grass. The cows move around randomly and eat grass. Consumed grass will regrow after some random time (but below a certain maximum). Cows compete thus for the grass, so some transactions may conflict and have to roll back.
3. The benchmark *Termites* has N termites. Each patch may contain 1 wood chip. Termites navigate randomly to find a wood chip, pick it up and move it next to other wood chip(s). Later on, sparse areas of wood chips are formed. In this benchmark, termites compete both for picking up and placing of wood chips.
4. The benchmark *Dummy* has N turtles, each simply wiggles randomly and moves. Similar to RedBlue, agents do not conflict, but furthermore they do not interact.

These benchmarks are run on a system provided by the SURF foundation with 32 cores Intel® Xeon E5-2698, 128 GB RAM. Hyper-Threading is disabled since it does not provide true CPU-core parallelism. The OS Ubuntu 16.04 64 bit was installed, with The Glorious Glasgow Haskell Compiler version 8.0.1 and NetLogo 5.3.1 running Java-8 version OpenJDK 1.8.0_111.

We run them on different configurations, with varying number of CPU cores (1, 2, 4, 8, 16, 32) and the problem size (N). 20 simulations are run for each benchmark and each configuration where we compute the average run time and resident memory. Note that NetLogo first needs to parse and compile the model, and then launches the JVM before it can start running a simulation. Being an embedded DSL, HLogo does not do these. To make the comparison fair, when measuring NetLogo's run time we *exclude* the time it takes to do the aforementioned preparation tasks. The benchmarks results are shown in Table 1: speedup is measured as the ratio of NetLogo execution time over the HLogo execution time for all experiments conducted; additionally, the memory ratio, i.e. HLogo memory used over the NetLogo memory used is given. Figure 4 shows a visualization of the absolute execution time of all four benchmarks.

Table 1. Execution speedup and memory usage of HLogo versions compared to NetLogo for a varying number of processors (cores). N is the size of the problem.

Problem	N	Execution speedup HLogo, #cores						Memory ratio HLogo, #cores					
		1	2	4	8	16	32	1	2	4	8	16	32
RedBlue	1000	0.74	1.18	1.82	2.87	**2.87**	1.33	0.12	0.19	0.3	0.34	0.38	0.44
	2500	0.55	0.67	1.04	1.96	**2.57**	2.33	0.12	0.18	0.25	0.48	0.72	0.72
	5000	0.56	0.65	0.99	1.77	3.01	**3.12**	0.12	0.18	0.27	0.47	0.85	0.98
	10000	0.54	0.67	1.09	2.02	3.24	**4.15**	0.13	0.19	0.28	0.48	0.85	*1.24*
	20000	0.46	0.53	0.82	1.73	3.22	**3.55**	0.16	0.21	0.33	0.48	0.91	*1.69*
	30000	0.40	0.50	1.00	1.82	3.23	**4.71**	0.18	0.26	0.39	0.65	*1.14*	*1.85*
Cows	100	3.29	5.08	7.57	**10.79**	8.99	6.54	0.10	0.17	0.3	0.52	0.82	0.87
	250	2.73	4.00	6.64	**8.57**	8.27	5.97	0.10	0.16	0.29	0.51	0.78	0.88
	500	2.04	3.36	5.19	6.90	**7.09**	6.55	0.10	0.17	0.3	0.54	0.87	*1.03*
	1000	1.55	2.59	4.08	5.37	**6.03**	5.97	0.10	0.16	0.29	0.53	0.88	*1.15*
	2000	1.16	1.84	3.27	4.52	5.28	**5.41**	0.10	0.17	0.29	0.53	0.92	*1.32*
	3000	0.90	1.04	2.28	3.57	4.86	**4.92**	0.10	0.17	0.25	0.46	0.91	*1.77*
Termites	100	1.26	1.72	2.45	**4.29**	3.96	3.03	0.10	0.16	0.26	0.37	0.39	0.43
	250	1.08	1.75	2.69	3.80	5.01	**5.01**	0.09	0.16	0.31	0.51	0.81	0.87
	500	1.09	2.06	3.15	5.17	**7.13**	6.42	0.10	0.17	0.31	0.58	*1.10*	*1.77*
	1000	0.58	1.05	2.04	2.98	4.35	**5.37**	0.10	0.16	0.3	0.57	*1.08*	*2.06*
	2000	0.42	0.84	1.45	2.78	4.61	**6.01**	0.09	0.14	0.27	0.51	0.97	*1.89*
	3000	0.46	0.67	1.17	2.30	4.03	**6.30**	0.09	0.15	0.25	0.48	0.90	*1.5*
Dummy	1000	0.54	0.95	1.24	1.56	**1.75**	1.17	0.25	0.45	0.84	*1.56*	*2.43*	*2.81*
	2500	0.45	0.55	0.90	1.59	**1.95**	1.79	0.22	0.3	0.49	0.99	*1.91*	*3.81*
	5000	0.41	0.51	0.91	1.60	**2.42**	2.38	0.23	0.34	0.52	0.96	*1.86*	*3.64*
	10000	0.40	0.48	0.85	1.58	2.77	**2.96**	0.24	0.32	0.56	0.91	*1.72*	*3.31*
	20000	0.33	0.42	0.70	1.30	2.52	**2.56**	0.28	0.41	0.58	0.99	*1.75*	*3.22*
	30000	0.32	0.53	0.94	1.60	2.39	**3.07**	0.3	0.44	0.68	*1.18*	*2.19*	*4.28*

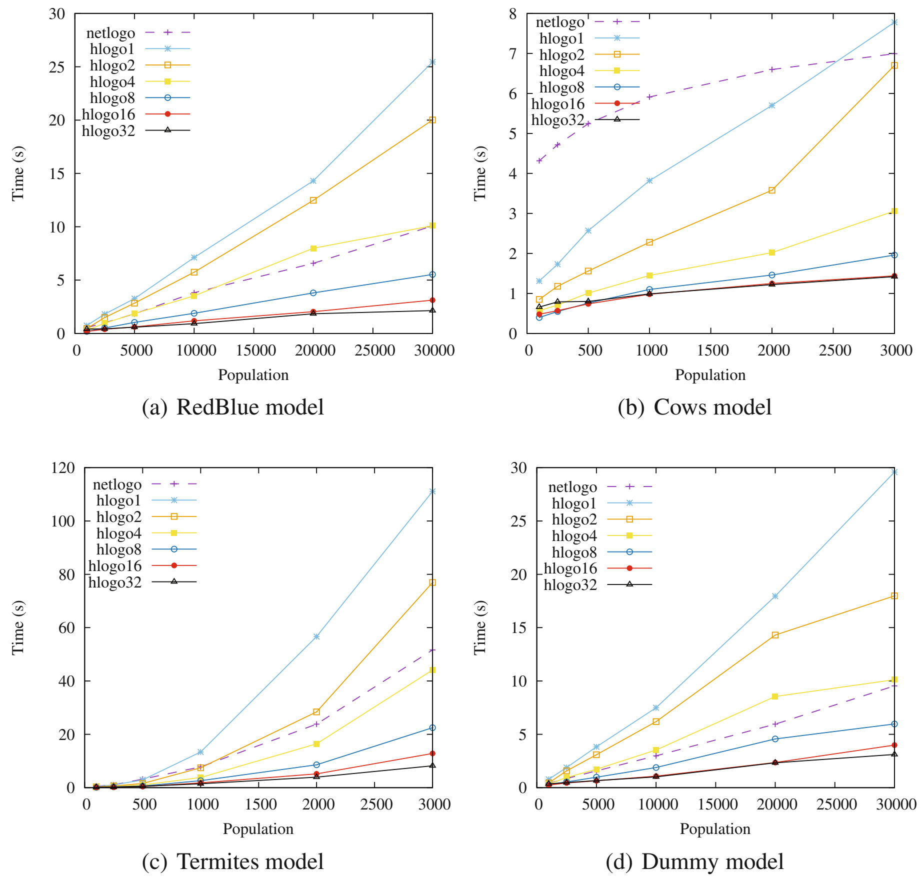

Fig. 4. NetLogo & HLogo execution time for the 4 benchmarks.

Overall, we can clearly witness the speed gain in HLogo as we increase the number of cores, while the performance scalability is retained. HLogo manages to be at its best 10.79 times faster than NetLogo using a configuration of 8 cores. Even with two cores HLogo manages to match or surpass the speed of NetLogo, while using on average 78% less memory, which is a positive outcome considering the fact that STM concurrency incurs certain overhead.

More specifically, the benchmark RedBlue at Fig. 4(a) shows linear growth of the execution time relative to the problem size, which is expected for this model. HLogo manages to be faster than NetLogo on a configuration of 4 cores and above; however, on less cores HLogo's speedup (see Table 1) actually worsens fast, attributed perhaps to the administrative costs of managing and distributing the `turtles` agentset to the different cores.

The results of the Cows benchmark (shown at Fig. 4(b)) indicate a sublinear complexity behaviour, since normally the overall workload grows less than the number of cows as there is no much grass left to be eaten. Again, the speedup of HLogo is positive for almost all core configurations; however, the speedup degrades (Table 1) as the problem size increases, which is due to increasing conflicts between the cows as they compete for the grass, hence leading to more STM rollbacks.

The Termites benchmark at Fig. 4(c) suggests a superlinear complexity of the model, since the turtles (termites) compete greatly with each other for (dis)placing the wood chips (patches) into forming piles. HLogo manages to be at its best 7.13 times faster than NetLogo using a configuration of 16 cores; the speedup degrades similarly to the Cows benchmark, attributed to the vast competition between the turtles.

The Dummy benchmark of Fig. 4(d) shows analogous complexity behaviour (linear) to the RedBlue benchmark. The HLogo speedup benefit is mostly positive but less than what is offered by RedBlue, although the problem has less agent interaction. This can be attributed to the fact Dummy's model has one large atomic block, whereas RedBlue has finer-grained atomic blocks. For the latter, this leads to smaller-sized STM logs and less traversal of the logs' contents when committing each STM log, i.e. atomically writing out the effects to memory.

HLogo, the above benchmarks, and other examples are available as open-source software at http://github.com/bezirg/hlogo.

8 Conclusions and Future Work

We have presented HLogo, a variant of the ABMS framework NetLogo, that offers an embedded DSL front end. At the back end it utilizes Software Transactional Memory (STM) to obtain performance gains from multi-core execution, and at the same time hide the complexity of concurrent programming from the programmers. As an embedded DSL, HLogo has the advantage of inheriting the Haskell's module system, thus allowing its programmers to import any of the whole wealth of Haskell libraries. Furthermore, the DSL is statically strongly typed for all its expressions and agent commands, which adds a certain level of safety when crafting a model.

Our benchmarks showed that HLogo's performance does not suffer from the use of STM. In fact, on multi CPU cores HLogo runs faster than NetLogo. Giving HLogo more cores progressively increases its speed. HLogo also uses less memory than NetLogo, up to a certain number of CPU cores (8 in most of our experiments' configurations). On the other hand, the trade off is that HLogo simulations on multiple cores are not reproducible. Despite this, we believe that there is room for applying HLogo, namely on problems where reproducibility is not a factor, and where speedup is crucial to keep the running time feasible.

As a final note, we want to add that HLogo's STM-based parallelism is in principle framework-agnostic and thus could be applied to other ABMS frameworks. For example, it can be applied to NetLogo. It should be possible to extend NetLogo with the `atomic` construct. Then, one can use one of available STM implementations in Scala (the implementation language of NetLogo) and mimic the described Haskell implementation. This might be attractive to the already large NetLogo community, thus giving NetLogo the performance benefits of HLogo.

Future Work. Our work so far has mainly focused on providing a front end ABMS DSL and its backend simulation engine. To improve HLogo's usability it will need a decent visualization front end, e.g. as NetLogo now has and this is a future work direction. Another feature that HLogo currently lacks and would come as a great addition is parameter sweeping, i.e. executing the model with multiple runs, each with a different parameter input (similar to NetLogo's BehaviorSpace tool).

With respect to its simulation engine, HLogo currently splits the workload to threads using a random divide and conquer strategy. If transactions that write to some common TVars are distributed to different CPU cores, this may lead to transactions conflicts and therefore rollbacks. On the other hand, assigning them to the same CPU core will avoid rollbacks. A smarter dynamic workload distribution strategy should take this into account. Such a strategy could be based on for example the turtles' last known positions, or how they are connected by links. Turtles that are close to each other, e.g. linked together, are more likely to conflict. A static approach is probably less likely to be successful because turtles move around, and links can be added and removed dynamically.

On the technical side, the turtles and links agentsets can be modified, currently, only in a non-concurrent fashion: a thread has to acquire a lock on the agentset to create or remove (`die`) a turtle or link. By changing the data structure to a concurrently modifiable tree-map implementation we would benefit from faster in-parallel insertions and deletions of turtles and links. Furthermore, we can consider optimizing the tree-map data structure used to store the turtles (see the beginning of Sect. 6) with a hybrid representation where turtles are added in-order into a `who`-indexed vector, which *if needed* is transformed into a sparse tree-map implementation when removals happen (as in 'die') dynamically at runtime. For the patches, we can consider low-level optimization of their array data structure by storing the indices (patches) in the so-called Z-order scan (similar to a zigzag) instead of linearly as it is now. A Z-order patch array would result in better performance because of better data locality: agents most often interact with 2D-neighbouring patches and would then store the data of the neighbouring patches close to each other, leading *on average* to less cache misses than accessing the array in the common row order.

HLogo currently requires programmers to specify the atomicity level of the agents by inserting calls to the function `atomic`. Through this mechanism, the programmers can increase parallelism e.g. by grouping an agent's access to unrelated TVars to different atomic blocks. As future work, we want to investigate if the function `atomic` can be extended e.g. to log information that would enable a simulation run to be reproduced or at least to some degree reconstructed, even if it was originally run on multiple cores. The challenge here is to be able to log enough information without slowing down the simulations. We also want to investigate if the insertions of `atomic` can be done automatically, e.g. through data flow analysis. Moreover, some STM transactions can be accelerated after applying certain optimizations, e.g. a wiggling cow move:

```
atomic( do{right =<< random 50 ; left =<< random 50})
```

can be optimized to `atomic( do{i ← random 50 ; j ← random 50 ; left(i+j)})` which is faster since the STM transaction log is shortened through combining two modifications of the turtle's heading (right, left) to one (left). The program remains consistent since this code runs atomically: no other agent could have, in anyway, witnessed the intermediate modification.

Finally, since Cloud computing has become widely available, it might also be interesting to investigate if HLogo can be extended to a distributed setting. For example, this would enable HLogo models to run in High-performance Computing (HPC). There is also the extreme case where models cannot fit in a single shared memory machine and have to be distributed to multiple processing nodes. There are Haskell technologies, such as Distributed Software Transactional Memory [20] or Cloud Haskell [12] that can be employed towards this direction.

Acknowledgements. This work was partially funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services (http://envisage-project.eu). All the benchmarks in this work were carried out on the Dutch national HPC e-infrastructure, kindly provided by the SURF Foundation (http://surf.nl).

References

1. Haskell, an advanced, purely functional programming language. https://www.haskell.org/
2. Grimm, V., et al.: A standard protocol for describing individual-based and agent-based models. Ecol. Modell. **198**(1–2), 115–126 (2006)
3. Berndsen, J.: The Hasim package. https://hackage.haskell.org/package/hasim
4. Bezirgiannis, N., Prasetya, I.S.W.B., Sakellariou, I.: HLogo: a parallel Haskell variant of NetLogo. In: Proceedings of the 6th International Conference on Simulation and Modeling Methodologies, Technologies and Applications, SIMULTECH, pp. 119–128. SciTePress (2016)
5. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: hardware design in Haskell. In: Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming. ACM (1998)
6. Castle, C.J.E., Crooks, A.T.: Principles and concepts of agent-based modelling for developing geospatial simulations, September 2006
7. Claessen, K., Pałka, M.H.: Splittable pseudorandom number generators using cryptographic hashing. In: ACM SIGPLAN Notices, vol. 48, pp. 47–58. ACM (2013)
8. Deissenberg, C., van der Hoog, S., Dawid, H.: EURACE: a massively parallel agent-based model of the European economy. Appl. Math. Comput. **204**(2), 541–552 (2008)
9. Discolo, A., Harris, T., Marlow, S., Peyton, Singh, S.: Lock-free data structures using STMs in Haskell, April 2006

10. D'Souza, R.M., Lysenko, M., Rahmani, K.: SugarScape on steroids: simulating over a million agents at interactive rates (2007)
11. Elliott, C.: Functional images. In: The Fun of Programming. Cornerstones of Computing. Palgrave, Basingstoke (2003)
12. Epstein, J., Black, A.P., Peyton-Jones, S.: Towards Haskell in the cloud. In: ACM SIGPLAN Notices, vol. 46, pp. 118–129. ACM (2011)
13. Epstein, J.M., Axtell, R.: Growing Artificial Societies: Social Science from the Bottom Up. Brookings Institution Press, Washington, D.C. (1996)
14. Gill, A.: Domain-specific languages and code synthesis using haskell. Queue **12**(4), 30 (2014)
15. Grimm, V., Revilla, E., Berger, U., Jeltsch, F., Mooij, W.M., Railsback, S.F., Thulke, H.H., Weiner, J., Wiegand, T., DeAngelis, D.L.: Pattern-oriented modeling of agent-based complex systems: lessons from ecology. Science **310**(5750), 987–991 (2005)
16. Hybinette, M., Kraemer, E., Xiong, Y., Matthews, G., Ahmed, J.: SASSY: a design for a scalable agent-based simulation system using a distributed discrete event infrastructure, pp. 926–933 (2006)
17. Kiran, M., Richmond, P., Holcombe, M., Chin, L. S., Worth, D., Greenough, C.: FLAME: simulating large populations of agents on parallel hardware architectures. In: International Foundation for Autonomous Agents and Multiagent Systems, pp. 1633–1636 (2010)
18. Knight, T.: An architecture for mostly functional languages. In: LFP 1986, pp. 105–112. ACM, New York (1986)
19. Koehler, M., Tivnan, B., Upton, S.: Clustered computing with Netlogo and RepastJ: beyond chewing gum and duct tape (2005)
20. Kupke, F.K.: Robust distributed software transactions for Haskell. Ph.D. thesis, Christian-Albrechts Universität Kiel (2010)
21. Logan, B., Theodoropoulos, G.: The distributed simulation of multiagent systems. Proc. IEEE **89**(2), 174–185 (2001)
22. Massaioli, F., Castiglione, F., Bernaschi, M.: OpenMP parallelization of agent-based models. Parallel Comput. **31**(10), 1066–1081 (2005)
23. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. **17**(3), 348–375 (1978)
24. North, M.J., Collier, N.T., Ozik, J., Tatara, E.R., Macal, C.M., Bragen, M., Sydelko, P.: Complex adaptive systems modeling with repast simphony. Complex Adapt. Syst. Model. **1**(1), 1–26 (2013)
25. Perfumo, C., Sönmez, N., Stipic, S., Unsal, O., Cristal, A., Harris, T., Valero, M.: The limits of software transactional memory (STM): dissecting Haskell STM applications on a many-core environment. In: CF 2008, pp. 67–78. ACM (2008)
26. Peterson, J., Hager, G.: Monadic robotics. In: Proceedings of the 2nd Conference on Domain-Specific Languages, DSL 1999, pp. 95–108. ACM, New York (1999)
27. Peyton Jones, S.L., Wadler, P.: Imperative functional programming. In: Proceedings of the 20th Symposium on Principles of Programming Languages, POPL. ACM (1993)
28. Pogson, M., Smallwood, R., Qwarnstrom, E., Holcombe, M.: Formal agent-based modelling of intracellular chemical interactions. Biosystems **85**(1), 37–45 (2006)
29. Railsback, S.F., Lytinen, S.L., Jackson, S.K.: Agent-based simulation platforms: review and development recommendations. SIMULATION **82**(9), 609–623 (2006)
30. Riley, P.F., Riley, G.F.: Next generation modeling III - agents: spades—a distributed agent simulation environment with software-in-the-loop execution. In: WSC 2003, Winter Simulation Conference, pp. 817–825 (2003)

31. Sakellariou, I., Kefalas, P., Stamatopoulou, I.: Enhancing NetLogo to simulate BDI communicating agents. In: Artificial Intelligence: Theories, Models and Applications. LNCS, vol. 5138, pp. 263–275. Springer (2008)
32. Salamon, T.: Design of Agent-Based Models. Eva & Tomas Bruckner Publishing, Repin (2011)
33. Shavit, N., Touitou, D.: Software transactional memory. In: PODC 1995, pp. 204–213. ACM (1995)
34. Sheard, T., Jones, S.P.: Template meta-programming for Haskell. SIGPLAN Not. **37**(12), 60–75 (2002)
35. Sorokin, D.: Aivika. http://www.aivikasoft.com/en/products/aivika.html
36. Tobias, R., Hofmann, C.: Evaluation of free java-libraries for social-scientific agent based simulation. J. Artif. Soc. Soc. Simul. **7**(1), 6 (2004)
37. Van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. Sigplan Not. **35**(6), 26–36 (2000)
38. Wilensky, U.: NetLogo (1999)
39. Wilensky, U.: Statistical mechanics for secondary school: the GasLab multi-agent modeling toolkit. Int. J. Comput. Math. Learn. **8**(1), 1–41 (2003)
40. Wilkerson-Jerde, M., Wilensky, U.: Restructuring change, interpreting changes: the deltatick modeling and analysis toolkit (2010)