Experience report: SoOSiM, an OS simulator in Haskell

Oleg Lobachev
Scuola Superiore Sant'Anna
Pisa, Italy
o.lobachev@sssup.it

Christiaan Baaij Jan Kuper
University of Twente
Twente, Netherlands
{c.p.r.baaij, j.kuper}@utwente.nl

Juri Lelli Giuseppe Lipari
Scuola Superiore Sant'Anna
Pisa, Italy
{j.lelli, g.lipari}@sssup.it

Abstract

This paper presents our experience from implementing SoOSiM—an operating system simulator, written in Haskell. We briefly discuss the goals and purposed usage of our simulator. We examine in detail the language features, extensions, programming patterns and techniques, used to implement SoOSiM. These include coroutines, type classes, monads, techniques for embedded domain-specific languages, and dynamic types. Coroutines were especially useful to implement synchronous communication model.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages

Keywords simulation, type classes, ad-hoc polymorphism, finally tagless embedding, dynamic typing, monads

1. Introduction

Simulation is often used to inspect various characteristics of a system. In the S(o)OS project[1] we investigate service-oriented aspects of operating systems. One part of the project requires to simulate the interactions between software components of an operating system (in the following: OS) and between OS and application software.

Common operating systems' designs have multiple drawbacks. The systems feature central synchronisation points for resource management and distribution; these hinder further scalability. Current commonly available OSes perform well on a handful of CPU cores, but they are not designed for thousands of cores! (However, we want to mention the Barrelfish research project [4] featuring advanced ideas for manycore architectures.) Heterogeneous (e. g., GPU+CPU, Cell BE), non-cache coherent architectures (e. g., Intel SCC) and network-on-a-chip approaches are quite poorly supported. The emerging cloud computing era requires much greater scalability and security than we saw before; this requires new OS design directions.

Overcoming the aforementioned limitations of contemporary operating system designs is the motivation for the S(o)OS project [34]. The project aims to research OS concepts and specific OS modules, which aid in scalability of the complete software stack (both OS and application) on future many-core systems. One of the key concepts of S(o)OS is that only those OS modules needed by an application thread, are actually loaded into the (local) memory of a CPU core on which the thread will run. This execution environment thus differs from contemporary operating systems where every core runs a complete copy of the (monolithic) operating system.

While creating new operating system concepts and regarding their interaction with the programmability of large-scale systems, we found that existing simulation packages do not seem to have the right abstractions for fast design exploration [2, 38]. The ability to simulate separate components of the OS and of the application was the main goal to develop the OS simulator SoOSiM [3]. The design decisions were:

- To simulate *message passing* and *shared memory* as the main forms of communication between software components.
- Achieve *synchronization* between components through *blocking* messaging.
- To have global *ticks*, designating an abstract notion of progress.
- To observe the number of ticks a software component is active, waiting for a response after invoking another component, or idle.

A note on blocking messaging: when a blocking message is sent, the sender waits for an answer—regardless *where* in the code of the sender the blocking *invoke* method is called. This means we required the notion of suspending a computation—it should be *suspendable* at any moment of execution of the simulated component. In the following we will show that using Haskell enabled us to implement this property in a nice way. The contributions of this paper include:

- A short overview of the SoOSiM simulator.
- The description of the interface used to implement OS components in SoOSiM.
- The description of *a* eDSL to describe applications that will run on a SoOSiM *simulated* system.
- The list of Haskell language features we found particularly useful in this project.

SoOSiM has been released on Hackage.[2]

The remaining part of this paper is organised as follows. Section 2 describes the structure of the SoOSiM simulator from the user's perspective. Section 3 gives an implementation overview of the domain specific language, used to implement application running within a SoOSiM simulated environment. Section 4 discusses Haskell features from which our implementation especially bene-

[1] See `http://www.soos-project.eu/` and [34] for details.

[2] Issue `cabal install SoOSiM` to install SoOSiM. See also: `http://github.com/christiaanb/SoOSiM`.

[Copyright notice will appear here once 'preprint' option is removed.]

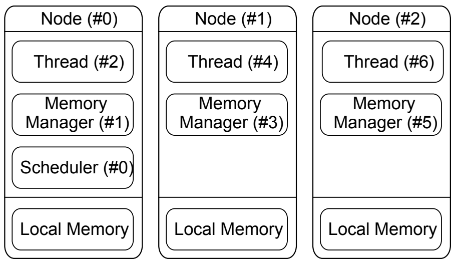

Figure 1. System, abstracted.

fited. Section 5 covers related work. Section 6 concludes and gives an outlook for the further research.

2. SoOSiM: An Overview

Basic structure. The purpose of SoOSiM is to provide a platform that allows a developer to observe the interactions between OS modules and application threads. For this reason the simulated hardware is highly abstract.

In SoOSiM, the hardware platform is described as a set of nodes. Each *node* represents a physical computing object: such as a core, a complete CPU, a memory controller, etc. Every node has a local memory of potentially infinite size. The layout and connectivity properties of the nodes are not part of the system description.

Each *node* hosts a set of components. A *component* represents an executable object: such as a thread, application, OS module, etc. Components communicate with each other either using direct messaging, or through the local memory of a node. Having both explicit messaging and shared memory, SoOSiM supports the two well known methods of communication. Components also have a message queue, because:

- Multiple components can potentially send messages to the same component concurrently.
- A component can receive messages while it is actually waiting for a response from another component.

All components in a simulated system, even those hosted within the same node, are executed concurrently, from the component's point of view. The simulator poses neither restrictions as to which components can communicate with each other nor to which node's local memory they can read from and write to. A schematic overview of an example system can be seen in Figure 1.

Agility. Basic requirements that we would have towards any simulator include the facilities to straightforwardly simulate the instantiation of application threads and OS modules. Aside from the fact that the S(o)OS-envisioned system will be dynamic as a result of loading OS modules on-the-fly, large-scale systems also tend to be dynamic in the sense that computing nodes can disappear (failure), or appear (hot-swap). Hence, we also require that our simulator facilitates the straightforward creation and destruction of computing elements, represented by *nodes* in SoOSiM. Our current need for a simulator rests mostly in formalising the S(o)OS concept, and examining the interaction between our envisioned OS modules and the application threads. As such, being able to extract highly accurate performance figures from a simulated system is not a key requirement. We do, however, wish to be able to observe all interactions among application threads and OS modules. Additionally, we wish to be able to *zoom in* on particular aspects of the behaviour of an application: such as memory access, messaging, etc. We detail on this in Section 3.

The simulator *attempts* to progress all components concurrently in one discrete step called a *tick*. If a component is waiting for a response from an invoked component, but none is available, a component will remain in a waiting state. During a tick, the simulator progresses a component by one of the following means:

- If a component is blocked on an invocation of another component, and a response is available in the message queue, the component will be executed with this response message.
- If a component is not blocked, and the message queue is not empty, the component will be executed by passing it the message that is at the head of the message queue.
- If a component is not blocked, and the message queue is empty, the component will be executed by passing a *null* message it.

When desired, a component can inform the simulator that it does not want to receive these null messages. In that case the component will not be executed by the simulator during a *tick*.

The simulator keeps several statistics with regards to the components, including how many ticks a component was either: active, waiting, or idle. When a component makes progress during a tick, it is considered *active*. When a component does not make progress because it is blocked on an invocation, and no response was available, the component is designated as *waiting*. A component not making progress because it has an empty message queue, and having indicated that it does not want to receive *null* messages, is designated as *idle*.

2.1 OS Component Descriptions

Like the simulator itself, the OS components are also specified in Haskell. Every component is modelled as a function. In case of SoOSiM, such a function is executed within the context of the simulator, this means: in a *monad*.

Because the function is executed within the monad, it can have *side-effects* such as sending messages to other components, or reading the local memory of a node. In addition, the function can be temporarily suspended at (almost) any point in the code. SoOSiM needs to be able to suspend the execution of a function such that we are able to emulate synchronous message passing between components, a subject further elaborated later on.

We describe a component as a function that receives a user-defined internal state as its first argument and a value of type $SimEvent$ as its second argument. The result of this function is the internal state. A value of type $SimEvent$ is either a message from another component or a null message:

$$\begin{aligned}&\mathbf{data}\ \mathit{SimEvent}\\&\quad = \mathit{Message}\ \mathit{ComponentId}\ \mathit{Dynamic}\\&\quad \mid \mathit{Null}\end{aligned}$$

We thus have the following *general* type signature for a component:

$$\mathit{component} :: \mathit{State} \rightarrow \mathit{SimEvent} \rightarrow \mathit{SimM}\ \mathit{State}$$

The simulator monad $SimM$ is described in Figure 4 in Section 4.3. The user-defined internal $state$ can be used to store any information that needs to perpetuate across simulator ticks. The type $State$ should be read as a place holder for any (user-defined) data type; there is no actual predefined $State$ data type.

To include a component description in the simulator, the developer will have to create an instance of the $ComponentIface$ *type class*. The $ComponentIface$ requires the instantiation of the following values to completely define a component:

- The initial internal state of the component.
- The unique name of the component.
- The monadic function describing the behaviour of the component.

We stress again that we are aiming at a high level of abstraction for the behavioural descriptions of our OS modules, where the focus is mainly on the interaction with other OS modules and application threads.

2.2 Interaction with the Simulator

Components have several functions at their disposal to interact with the simulator and consequently interact with other components. The available functions are:

- *createComponent* instantiates a new component on a specified node.
- *invoke* sends a message to another component and waits for the answer. Whenever a component uses this function it will be temporarily suspended by the simulator. Several simulator ticks might pass before the response. Once the response is available the simulator resumes the execution of the calling component.
- *invokeAsync* sends a message to another component and register a handler with the simulator to process the response. In a contrast to *invoke*, using this function will *not* suspend the execution of the component.
- *respond* sends a message to another component as a response to an invocation.
- *yield* informs the simulator that the component does not want to receive null messages.
- *readMem* performs a read at a specified address of a node's local memory.
- *writeMem* writes a value at a specified address of a node's local memory.
- The function *componentLookup* performs a lookup of the unique identifier of a component on a specified node.

Components have two unique identifiers, their global *name* (as specified in the *ComponentIface* instance), and a *ComponentId* that is a unique number corresponding to a specific instance of a component. When a programmer wants to *invoke* a component, she needs to know the unique *ComponentId* of the specific instance. To give a concrete example, using the system from Figure 1 as our context: *Thread (#6)* wants to invoke the instance of the *Memory Manager* that is running on the same Node (#2). As *Thread (#6)* was not involved with the instantiation of that OS module, it has no idea what the specific *ComponentId* of the memory manager on Node #2 is. It knows the unique global name of the memory manager in question, so it can use the *componentLookup* function to find the *Memory Manager* with ID #5 that is running on Node #2.

2.3 Synchronisation between Modules

A *naive* approach to model synchronisation between OS modules is to implement an OS module as a finite state machine (FSM). In state 'X' OS module 'Bob' could send a message to 'Alice' and enter state 'Y'. Only when receiving a response from 'Alice', would 'Bob' progress from state 'Y' to state 'Z'. If we wanted, we could even have implemented OS modules as *pure* functions, which, aside from returning an updated state, would also return a simulator event. One such event would then be sending a message to another component.

When sending a message the FSM would enter a new state for that specific invocation of another component. As a result, designers of OS module descriptions would have to extend the dictionary of states for *every* invocation their module makes. As one can imagine, the explicit FSM approach would quickly result in overly verbose and unstructured OS module descriptions.

As a solution we essentially mechanised the creation of the FSM using the notion of *suspendable* computations. When a developer uses the *invoke* function, two things actually happen:

- A message is send to the invoked component (callee).

```
class Symantics repr where
  lam  :: (repr a → repr b) → repr (a ↣ b)
  app  :: repr (a ↣ b) → repr a → repr b
  drf  :: repr (Ref a) → repr a
  (=:) :: repr (Ref a) → repr a → repr Void
```

Figure 2. An embedded language, a fragment. Viz. [9]

- The remaining computation of the caller is suspended and stored by simulator, ready to be scheduled once a response to the invocation arrives.

Every call to *invoke* hence corresponds to an explicit state transition, but without the burden of explicit state-keeping. We achieved suspendability using coroutines [10], more specifically, a coroutine monad transformer [8]. We return to the subject of the coroutine monad transformer in Section 4.

3. A Domain-Specific Language for SoOSiM

One of the purposes of SoOSiM is to observe the interaction between application and OS. We however do not want to *pollute* our application descriptions with calls to the simulator functions. Nor do we want to be forced to describe our application in a monadic style. Additionally, we want to be in control which interactions between OS and application are exactly simulated, without having to change either the descriptions of the OS modules or the application when we want to observe different aspects.

For the above reasons we have decided to make use of embedded languages for the definitions of our applications. The idea is to specify an application once in a self-created embedded language and define different interpretations for the embedded language constructs as the observations of the interaction between application and OS [16].

Following the *final tagless* [9] encoding of embedded languages in Haskell, we use a type class to define the language constructs. As an example we demonstrate a mini functional language with mutable references. A partial specification of the *Symantics* (a pun on *syntax* and *semantics* [9]) type class, defining an *embedded language*, is shown in Figure 2. The arrow $\rightarrowtail$ is the function arrow in the embedded language, e. g., $repr\ (a \rightarrowtail b)$, in contrast to the function arrow $\rightarrow$ of the host language.

One interaction we might now want to observe is the application's use of mutable references and the subsequent communication with the memory manager OS module. For this purpose we create a **newtype** *RefMemAcc* wrapping the simulator monad, and define an instance of the *Symantics* type-class for it.

We can define most language constructs to behave like their Haskell counterpart, e.g., the *app* construct instance is defined using monadic bind ($\gg\!=$). However, for the definition of the mutable reference aspect of our embedded language, we implement these constructs as invocations of the memory manager (see Figure 3), instead of using a Haskell-based implementation like `Data.IORef`.

Final tagless embedding. Carette et al. have demonstrated how a typed interpreter of a typed language can be tagless [9]. This approach does not use GADTs or dependent types. The target language types are represented as host language types, object terms are encoded as calls to host language functions. With type classes Carette et al. make it possible to express, e. g., an interpreter and an compiler of the same language in a flexible manner. Hofer et al. do a similar work [16], but focus on different interpretations. This matches our intentions. We use our above example of a small language with mutable references. If we want to observe memory interactions in a more detail, we embrace an additional implementation of the language,

```
newtype RefMemAcc = RMA SimM
instance Symantics RefMemAcc where
  drf x = RMA $ do
    i     ← foo x
    mmId ← componentLookup "MemoryManager"
    fmap unmashal $ invoke mmId (marshal (Read i))
  x =: y = RMA $ do
    i     ← foo x
    v     ← bar y
    mmId ← componentLookup "MemoryManager"
    fmap unmashal $ invoke mmId (mashal (Write i v))
```

Figure 3. Observing memory access (partial definition).

```
type SimMonad = StateT SimState IO
data SimState   = ...
newtype SimM a
  = SimM {runSimM ::
      Coroutine
        (RequestOrYield ComponentID Dynamic)
        SimMonad
        a
    } deriving (Functor, Monad)
data RequestOrYield request response x
  = Request request (response → x)
  | Yield x
instance Functor (RequestOrYield x f) where
  fmap f (Request x g) = Request x (f ∘ g)
  fmap f (Yield y)      = Yield (f y)
```

Figure 4. Implementing *SimM*.

where the functions for the memory interaction are exactly studied and basically ignore the observation of other parts of the language. In a similar manner we can introduce further implementation of the same language with different properties.

4. Features We Used

4.1 Existential Types

We employ existential types to store the differently typed states of all the components running in our simulated environment. Components are subsequently implemented as type class instances, giving us a straightforward way to access the existentially quantified values in our collection of states.

4.2 Type Classes

Type classes are naturally used for the eDSL encoding from the previous section [9, 12].

Beyond this, we use type classes to express many other useful abstractions. For instance, an *interface* for the component of the OS is a type class. Thus, each interacting 'building block' in the simulated software is an instance of that particular type class. Although initially used solely for the purpose of dealing with existentially quantified values, this *ComponentIface* type class enables great flexibility, including an option to extend the kinds of simulated objects by third party.

4.3 Monads

We use a monad called *SimM* to capture the *state* of the simulator. The implementation of *SimM*, sketched in Figure 4, enables us to reach the main implementation goal: the ultimate suspension and resumption of the components upon message passing. To suspend a computation in the current component we can now merely write *request componentId*, where the *componentId* is the unique ID of some other component. The suspending of a component is nothing else than sending a outgoing message in the name of the component and blocking the sender until a response. To continue the execution of a resumeable computation we issue *resume computation*.

4.4 Coroutines

As we mentioned above and as one could infer from Figure 4, the implementation makes use of the coroutines [10]. With this concept we capture the notion of multiple suspending and resuming computations. The need in such a behaviour arises naturally with the blocking message passing between components. We utilise Control.Monad.Coroutine [8] to express coroutines.

4.5 Dynamic Types

We use dynamic types [1] as 'the other side' of the type classes: we need to communicate heterogeneous types over a typed channel, and there is no option to know all the types-to-transmit at the moment of the channel definition.

The *origin* for the need of dynamic types lies in the suspension functor of our coroutines. More specifically, the *response* field, that designates the type of the response returned after an invocation (see Figure 4). This field would be different for every component we want to invoke, as components should be able to communicate data of any type. We have only been able to encode this aspect in the suspension functor using dynamic types.

If we knew upfront what kind of operations the users might want to perform on the data types, we could have used existential types and type classes, *viz.* [22, 27]. However, this is not our case. We need to synchronously transmit values of heterogeneous types. The types in the coroutine monad transformer are too restrictive. Trying to define a type class or a sum type for this setting would yield either a too large, or a too restrictive definition. We utilised Data.Dynamic to convert transmitted values of a dynamic type into values of concrete monomorphic types. An attempt to sort the dynamic types out with the parameterised monad also fails, as detailed in the next section.

4.6 Parameterised Monads

The parameterised monads[3] allow more generic type signatures for monad operations, esp. for ≫= [24]:

```
class PMonad m where
  return :: a → m s s a
  (≫=) :: m s1 s2 a → (a → m s2 s3 b) → m s1 s3 b
```

If we could switch from one intermediate type to another in the coroutine, we would need no dynamic types. This approach works fine with the state monad

```
newtype PState s1 s2 a
  = PState {runPState :: s1 → (a, s2)}
instance PMonad PState where
  return a = PState $ λs → (a, s)
  m ≫= k  = PState $ λs →
                let (a, s′) = runState m s
                in runState (k a) s′
```

[3] See http://www.haskell.org/pipermail/haskell-prime/2006-February/000498.html.

but it fails for the coroutines. The pivotal issue is: if we introduce different types for the intermediate states of the coroutine, we end up adding more and more types — the coroutine definition is recursive. In this (non proper Haskell) code

```
newtype Coroutine m o i r
  = Coroutine {resume ::
        m (Either
            (o, i → Coroutine m i2 o r)
            r
        )}
```

we make an attempt to reach our goal. Let us abbreviate left and right hand side of the above equation with LHS and RHS correspondingly. However, to make this code valid Haskell, we need to reference *i2* on the LHS: *Coroutine m i i2 o r*. In the type of the *resume* function on the RHS we need to reference *Coroutine* correctly, hence we write *Coroutine m i2 i3 o r* there. We need to add *i3* on the LHS, and so on. Using some more advanced tricks, like a type class for all the chain *i*, *i2*, *i3*, ... does not really help: all the types on the RHS still need to be listed on the LHS, thus exploding the definition.

5. Related Work

Operating systems and simulators. COTSon [2] is a full system simulator that allows a developer to execute normal x86 code in a simulated environment. COTSon is far too detailed for our needs, and does not facilitate the easy exploration of a complete operating system.

OMNeT++ [38] is a C++-based discrete event simulator with focus on parallel systems. OMNeT++ is too static for our purposes, it disallows dynamic modules.

Hallgren et al. describe a basic operating system implementation in Haskell called House [13]. This work is in a sense dual to ours: we simulate an OS. Hallgren et al. modified GHC run-time system to allow code execution on bare metal. In House, OS modules are executed within the *Hardware* monad, allowing direct interaction with real hardware. This approach is comparable with our *SimM* monad. However, as we are more concerned with interaction of OS modules, House in not suitable for our purposes: OS modules in House must be implemented in full detail.

Barrelfish [4] is an OS in which domain-specific languages are used, amongst other purposes, to define driver interfaces [11]. These embedded languages are also implemented in Haskell. The approach used in Barrelfish is however to create parsers for their languages, in a contrast to our embedding approach.

Embedded domain-specific languages. Embedding programming languages was proposed by Landin [25]. For the concept of an eDSL see the papers by Hudak [17, 18], the 1998 paper describes pure embedding. The literature on this topic is quite broad, an overview is presented, e. g., by Van Deursen et al. [37].

The final tagless embedding of DSLs originates from the seminal paper by Carette et al. [9]. Hofer et al. [16] presented a paper that compared to the work by Carette et al. [9] is more composable, allows subtyping, and is done in Scala [31]. These two papers have different goals: Carette et al. implement DSLs for higher-order languages, while Hofer et al. focus on the possibility to plug in multiple interpretations of a pure embedded DSL. The latter corresponds more with our goals: we want to control what we can observe in the particular instance.

Suspendable computation. We used coroutine monad transformer [8] to express a suspendable computation. Other approaches to this include the suspension monad implementations of iteratees/enumeratees. This approach was introduced in [20], see, e. g., an overview by Lato [26]. See also Oleg Kiselyov's new paper [21] for the current description of iteratee-based IO in Haskell.

Haskell experience reports. Haskell has been successfully used to implement a range of applications far from typical programming language research. This is not surprising for a general purpose language, but nevertheless we would like to list most interesting applications. We already mentioned House [13] in the previous section.

Haskell has been used to facilitate a formal verification of a microkernel [23], to implement commercial mission-critical applications [33], to manage a Linux distribution [5], to configure realtime OS components [19], etc.

6. Conclusions and Future Work

Conclusions. We have presented the efforts required to implement SoOSiM, an OS simulator in Haskell. We note that the techniques more commonly used in the programming language research were highly applicable for our purposes.

We have demonstrated how the utilisation of final tagless eDSL construction [9, 16], type classes [12], monads [39], *Dynamic* types [1], and coroutines [8, 10] facilitated an abstract and concise implementation of an operating system simulator. It is uncommon to use all these features for such a task — a fact that emphasises the novelty of our work.

Real life simulations. The major goal of the project is to simulate the behaviour of a real life application and to draw conclusions therefrom. We are currently actively researching these issues.

Concurrency. One aspect of the future work is the concurrent SoOSiM. One option is to use Concurrent Haskell [32], it provides (concurrent) green threads. However, it would also require to use software transactional memory [35] because of multiple writes to the same data. Examples of fruitful combinations of this concept with other ones include papers by Harris et al. [14] and Bieniusa et al. [6, 7]. Another option would be to use a *parallel* Haskell like Multicore Haskell [29], *Par* monad [30] or Eden [28]. The 'multiple writes' issue, however, needs some further handling in this case. Two further options are either to make the coroutine execution concurrent using actors [15, 36] or to switch to iteratees for expressing blocking communication and to utilise transformers from usual to parallel composable iteratees[4].

Acknowledgments

This work was supported by the S(o)OS project, sponsored by the European Commission under FP7-ICT-2009.8.1, Grant Agreement No. 248465. We thank Chung-chieh Shan and Oleg Kiselyov for valuable remarks on *Dynamic* types and the parallel iteratees correspondingly. We also thank Tommaso Cucinotta for a fruitful discussion.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.*, 13(2): 237–268, Apr. 1991. ISSN 0164-0925. DOI 10.1145/103135.103138.

[2] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. COTSon: Infrastructure for full system simulation. *SIGOPS Oper. Syst. Rev.*, 43(1):52–61, Jan. 2009. ISSN 0163-5980. DOI 10.1145/1496909.1496921.

[4] See `http://projects.haskell.org/pipermail/iteratee/2011-July/000083.html` for details.

[3] C. Baaij, J. Kuper, and L. Schubert. SoOSiM: Operating system and programming language exploration. In *3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, WATERS '12, July 2012.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44. ACM, 2009. ISBN 978-1-60558-752-3. DOI 10.1145/1629575.1629579.

[5] C. Beshers, D. Fox, and J. Shaw. Experience report: using functional programming to manage a Linux distribution. *SIGPLAN Notices*, 42(9):213–218, Oct. 2007. ISSN 0362-1340. DOI 10.1145/1291220.1291184.

[6] A. Bieniusa and T. Fuhrmann. Lifting the barriers — reducing latencies with transparent transactional memory. In L. Bononi, A. Datta, S. Devismes, and A. Misra, editors, *Distributed Computing and Networking*, LNCS 7129, pages 16–30. Springer, 2012. ISBN 978-3-642-25958-6. DOI 10.1007/978-3-642-25959-3_2.

[7] A. Bieniusa, A. Middelkoop, and P. Thiemann. Brief announcement: actions in the twilight — concurrent irrevocable transactions and inconsistency repair. In *Proceeding of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 71–72. ACM, 2010. ISBN 978-1-60558-888-9. DOI 10.1145/1835698.1835714.

[8] M. Blažević. The monad-coroutine package. Hackage, 2012. `http://hackage.haskell.org/package/monad-coroutine`. Retrieved 23.5.2012.

[9] J. Carette, O. Kiselyov, and C.-c. Shan. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *Journal of Functional Programming*, 19(5):509–543, Sept. 2009. ISSN 0956-7968. DOI 10.1017/S0956796809007205.

[10] M. E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, July 1963. ISSN 0001-0782. DOI 10.1145/366663.366704.

[11] P.-E. Dagand, A. Baumann, and T. Roscoe. Filet-o-Fish: practical and dependable domain-specific languages for OS development. *SIGOPS Oper. Syst. Rev.*, 43(4):35–39, Jan. 2010. ISSN 0163-5980. DOI 10.1145/1713254.1713263.

[12] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, Mar. 1996. ISSN 0164-0925. DOI 10.1145/227699.227700.

[13] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach. A principled approach to operating system construction in Haskell. *SIGPLAN Notices*, 40(9):116–128, Sept. 2005. ISSN 0362-1340. DOI 10.1145/1090189.1086380.

[14] T. Harris, S. Marlow, S. P. J. Jones, and M. Herlihy. Composable memory transactions. *Comm. ACM*, 51:91–100, 2008. ISSN 0001-0782. DOI 10.1145/1378704.1378725.

[15] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245. Morgan Kaufmann, 1973.

[16] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, GPCE '08, pages 137–148. ACM, 2008. ISBN 978-1-60558-267-2. DOI 10.1145/1449913.1449935.

[17] P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es), Dec. 1996. ISSN 0360-0300. DOI 10.1145/242224.242477.

[18] P. Hudak. Modular domain specific languages and tools. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 134–142, June 1998. DOI 10.1109/ICSR.1998.685738.

[19] M. P. Jones. Experience report: playing the DSL card. *SIGPLAN Notices*, 43(9):87–90, Sept. 2008. ISSN 0362-1340. DOI 10.1145/1411203.1411219.

[20] O. Kiselyov. Incremental multi-level input processing with left-fold enumerator: predictable, high-performance, safe, and elegant. In *ACM SIGPLAN Developer Tracks on Functional Programming*, DEFUN '08, 2008.

[21] O. Kiselyov. Iteratees. In T. Schrijvers and P. Thiemann, editors, *Functional and Logic Programming*, LNCS 7294, pages 166–181. Springer-Verlag, 2012. ISBN 978-3-642-29821-9. DOI 10.1007/978-3-642-29822-6_15. See `http://okmij.org/ftp/Haskell/Iteratee/describe.pdf` for an extended version.

[22] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 96–107. ACM, 2004. ISBN 1-58113-850-4. DOI 10.1145/1017472.1017488.

[23] G. Klein, P. Derrin, and K. Elphinstone. Experience report: sel4: formally verifying a high-performance microkernel. *SIGPLAN Notices*, 44(9):91–96, Aug. 2009. ISSN 0362-1340. DOI 10.1145/1631687.1596566.

[24] E. Kmett and D. Devriese. The monad-param package. Hackage, 2012. `http://hackage.haskell.org/package/monad-param`. Retrieved 30.5.2012.

[25] P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, Mar. 1966. ISSN 0001-0782. DOI 10.1145/365230.365257.

[26] J. W. Lato. Iteratee: Teaching an old fold new tricks. *The Monad. Reader*, (16):19–35, 2010.

[27] K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *ACM Trans. Program. Lang. Syst.*, 16(5):1411–1430, Sept. 1994. ISSN 0164-0925. DOI 10.1145/186025.186031.

[28] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel functional programming in Eden. *Journal of Functional Programming*, 15(3): 431–475, 2005.

[29] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore Haskell. *ACM SIGPLAN Notices*, 44(9):65–78, 2009.

[30] S. Marlow, R. Newton, and S. Peyton Jones. A monad for deterministic parallelism. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 71–82. ACM, 2011. ISBN 978-1-4503-0860-1. DOI 10.1145/2034675.2034685.

[31] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Inc, 2008. ISBN 0981531601.

[32] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 295–308. ACM, 1996. ISBN 0-89791-769-3. DOI 10.1145/237721.237794.

[33] C. J. Sampson. Experience report: Haskell in the 'real world': writing a commercial application in a lazy functional lanuage. *SIGPLAN Notices*, 44(9):185–190, Aug. 2009. ISSN 0362-1340. DOI 10.1145/1631687.1596578.

[34] L. Schubert, A. Kipp, B. Koller, and S. Wesner. Service-oriented operating systems: future workspaces. *Wireless Communications, IEEE*, 16(3):42–50, june 2009. ISSN 1536-1284. DOI 10.1109/MWC.2009.5109463.

[35] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10:99–116, 1997. ISSN 0178-2770. DOI 10.1007/s004460050028.

[36] M. Sulzmann, E. S. L. Lam, and P. Van Weert. Actors with multi-headed message receive patterns. In *Proceedings of the 10th International Conference on Coordination Models and Languages*, pages 315–330. Springer, 2008. ISBN 3540682643.

[37] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[38] A. Varga and R. Hornig. An overview of the OMNeT++ simulation environment. In *Proceedings of Simutools '08*, pages 1–10, ICST, Brussels, Belgium, 2008. ISBN 978-963-9799-20-2.

[39] P. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 61–78. ACM, 1990. ISBN 0-89791-368-X. DOI 10.1145/91556.91592.