Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 255 (2009) 45–64
www.elsevier.com/locate/entcs

Implementation of an Orchestration Language as a Haskell Domain Specific Language

Marco Devesas Campos[1]

Nordic DataGrid Facility
Copenhagen, Denmark

L. S. Barbosa[2,3]

DI-CCTC
Universidade do Minho
Braga, Portugal

Abstract

Even though concurrent programming has been a hot topic of discussion in Computer Science for the past 30 years, the community has yet to settle on a, or a few standard approaches to implement concurrent programs. But as more and more cores inhabit our CPUs and more and more services are made available on the web the problem of coordinating different tasks becomes increasingly relevant.
The present paper addresses this problem with an implementation of the orchestration language Orc as a domain specific language in Haskell. Orc was, therefore, realized as a combinator library using the lightweight threads and the communication and synchronization primitives of the Concurrent Haskell library. With this implementation it becomes possible to create orchestrations that re-use existing Haskell code and, conversely, re-use orchestrations inside other Haskell programs.
The complexity inherent to distributed computation, entails the need for the classification of efficient, re-usable, concurrent programming patterns. The paper discusses how the calculus of recursive schemes used in the derivation of functional programs, scales up to a distributed setting. It is shown, in particular, how to parallelize the entire class of binary tree hylomorphisms.

Keywords: Orc, Coordination Languages, Haskell, Thread-based Programming, Parallel Divide-and-Conquer Algorithms

1 Introduction

While two decades ago the expression *programming-in-the-large* was coined to emphasise modular decomposition in software design, *programming-in-the-world* emerged more recently to cater for radically new challenges placed on the nature of

[1] Email: devesas.campos@gmail.com
[2] Email: lsb@di.uminho.pt
[3] Acknowledgment: This research was partially supported by FCT (the Portuguese Foundation for Science and Technology), under contract PTDC/EIA/73252/2006.

doi:10.1016/j.entcs.2009.10.024

software composition. Actually, at present, the point is not only to master the complexity of building and deploying a large application in time and budget, but also to manage an open-ended structure of autonomous components, possibly distributed and highly heterogeneous. The ubiquity of concurrency, in the double perspective of distribution and parallelism, entails the need for new approaches and languages for composing, at runtime, interacting software.

The coordination paradigm [8,17,1], claiming for a strict separation between effective computation and its control is one such answer. It emerged from the need to exploit the full potential of massively parallel systems, which requires models able to deal, in an explicit way, with the concurrent execution and cooperation among very large number of heterogeneous, autonomous and loosely-coupled components. Coordination models make a clear distinction between such components and their interactions, and concentrate on their joint *emergent* behaviour. Synchronisation, communication, reconfiguration, creation and termination of computational activities are, thus, primary issues of concern. Recent languages, such as Reo [2] and Orc [12,16,22], provide semantically sound frameworks for specifying coordination strategies and protocols, as well as for implementing and reasoning about such specifications.

The present paper is a step in this direction: it discusses the implementation of Orc as as a Haskell [10] domain specific language, to which we gave the unimaginative name of HOrc — available at `http://wiki.di.uminho.pt/twiki/bin/view/Research/HOrc/WebHome`. Developed as a combinator library, HOrc programs are valid Haskell functions, executable by the Haskell runtime: there is no intermediate representation of programs, nor the need for a separate runtime.

Implemented as a domain specific language in Haskell, Orc becomes available to a vast community of users and developers. For example, a main motivation for developing HOrc was the possibility of coupling an Orc animator to CoorInspector, a tool for extracting coordination scripts (expressed in Orc) from legacy code [19]. HOrc makes possible to validate such scripts and, eventually, to transform them.

Clearly, Haskell's expressiveness, higher-order functions, extensibility and associated libraries, make it an ideal means to support domain specific languages. In particular, the development of HOrc relied heavily on the *Concurrent Haskell* [11] native library, which provides the basic concurrency primitives used to code our combinators: thread manipulation, synchronization and communication primitives.

But Haskell is also associated to program calculus, in the spirit of [3], which has been extremely effective in classifying, expressing and calculating structural recursion patterns, such as catamorphisms (which encode inductive schemes) or hylomorphisms (which combine induction and coinduction). Although the impact of such a taxonomy has been enormous in the way software is developed, only a few authors, and usually out of the mainstream, such as [21] and [20], have studied how to explore the parallelism underlying such recursion schemes.

In this paper we show how to express in HOrc distributed versions of typical functional combinators: from elementary ones, such as *split* in which two well-

$$
\begin{array}{rcl}
Expr & ::= & SiteCall \\
& | & FuncCall \\
& | & Expr >Var> Expr \\
& | & Expr \mid Expr \\
& | & Expr \textbf{ where } Var :\in Expr
\end{array}
$$

$$
\begin{array}{rcl}
SiteCall & ::= & Site(Arg, Arg, \ldots) \\
FuncCall & ::= & Func(Arg, Arg, \ldots) \\
FuncDef & ::= & Func(Var, \ldots) = Expr
\end{array}
$$

$$
\begin{array}{rcl}
Arg & ::= & SiteName \\
& | & Var \\
& | & Lit
\end{array}
$$

$$
\begin{array}{rcl}
Lit & = & String \cup Int \cup \ldots \\
Var & = & name \\
Site & = & name \\
Func & = & name
\end{array}
$$

Table 1
The definition of Orc Expressions

typed computations run in parallel, to hylomorphisms that independently and in parallel invoke the recursive calls of processes. An example is provided in section 4. The remaining sections include a brief introduction to Orc, in section 2, and the development of HOrc as a domain specific language, the paper's main contribution, in section 3.

2 A Short Introduction to Orc

This section is a quick introduction to the language Orc as presented in [16] — the reader is referred there for more details. Orc's original aim was to permit simple implementation of programs that use web-services to obtain and process data. Such programs are characterized by being very loosely coupled, having a very weak set of possible assumptions. A language capable of dealing with such general cases — like Orc is — is also capable of expressing many distributed patterns.

Orc programs — or orchestrations — are expressed by Orc expressions. Expressions can be of one of three types: site calls, Orc operators and Orc functions. Orchestrations are built by cobbling simple expressions together using the operators to form increasingly more complex expressions. The result of an expression is the set of values that it publishes: an expression may publish zero, one or several values. The definition of the language can be found in table 1 [4]. Throughout the rest of this section we will present each component individually.

2.1 Site calls

The basic unit of an Orc program is the site call. Site calls encapsulate many different ways of processing values: functions, procedures, remote procedure calls (RPC), SOAP requests, remote method invocations (RMI), etc. Their syntax is similar to function calls in C: the name of the site followed by a tuple containing both variables and literals. Nested calls — e.g. $fib(pow(2, 4))$ — are not permitted, though.

The following examples present two site calls that we would expect correspond to different types of calls. $askUser$, would ask a person what he wants from Amazon —

[4] A small detail note: the definition of Orc includes also a special operator called 0. In HOrc, though, 0 was implemented as a primitive site. To keep the symmetry of the presentation, we will defer the introduction of 0 until the primitive sites.

typically through a form, or a prompt if he is old-school. This would be done locally while the call $buyFromAmazon$ would generate a remote request of an approriate type. Their syntax, though, is the same.

$$askUser('John\ Doe')$$
$$buyFromAmazon(gift\_variable)$$

Sites may return values that can be used by other site calls. We say that such values are published by the site call. Due to the need for generality, site calls have very loose semantics: sites may take arbitrary time to respond or even not at all. So each individual site call may publish zero or one value.

2.2 Sequential Composition

The first operator we will look into is sequential composition. Like composition in imperative languages, two sub-expressions — a preceding and a subsequent expression — are linked together by this operator and run in the order they appear. This operator also expresses a dependency of the subsequent sub-expression with respect to the values published by the preceding sub-expression. Its general form is

$$f >x> g(x)$$

In this expression, values published by f are assigned to variable x. g can use said publications by referencing x in its definition. Example:

$$askUser('JohnDoe') >x> buyFromAmazon(x)$$

In the example above, the preceding sub-expression is a simple site call, thus x can only take one value (at most). But in the general case multiple values may be be published — what should x stand for then? The first value? The second? The last? The answer is: all of them. For every publication of the preceding sub-expression, the sequential composition combinator starts the subsequent expression in an independent thread and assigns the value of x for that instance to the value just published.

If no value is published by the preceding sub-expression, then all references to x inside the subsequent sub-expression are undefined. Running the subsequent would lead, in general, to an error. So, even when the variable of the sequential composition is not referenced, the subsequent expression is only ran when the preceding sub-expression publishes a value.

Sequential composition acts like a quantifier in a formal language: it binds appearances of symbol x in g to the values published by the left sub-expression. Topics such as freeness of variables and scope of quantification — prevalent when we talk about formal languages — also apply. In the scope of this and the remainder operators we use the standard interpretation of those rules.

2.3 Synchronous Parallel Composition

If sequential composition introduces a total ordering on the evaluation of expressions, synchronous parallel composition permits simultaneous evaluation of expres-

sions. Given an Orc expression of the form

$$f \mid g$$

the evaluations of both f and g start immediately and in parallel.

The second role of this operator is to combine the publications of its sub-orchestrations — it forwards to the outside world every value that is published by either one of them. The following example does the shopping for an entire family:

$$\begin{array}{l}(\quad askUser('Dad') \mid askUser('Mom') \\ \quad \mid askUser('Son') \mid askUser('Daughter') \ ) >x> buyFromAmazon(x)\end{array}$$

Because the operator is associative, we have omitted some nested parenthesis in the definition. Also, notice the interaction of this operator and sequential composition: for every member of the family a different instance of $buyFromAmazon(x)$ will be initiated.

2.4 Asynchronous Parallel Composition

Orc's most distinguished feature is, arguably, the asynchronous parallel composition operator — a.k.a. **where**. This operator is a mixture of the two previous operators plus some idiosyncrasies of its own. Its general form is:

$$g(x) \textbf{ where } x :\in f$$

Like sequential composition, values published by f are assigned to references of the quantified variable in g. Unlike sequential composition, and much like synchronous parallel composition, an instance of f and an instance of g are started simultaneously by **where**.

Because they start at the same time, it is possible that x is needed by g before f has had time to publish a value. To accommodate for such cases, variable x is initially given a special, undefined value. When, during the invocation of a site call — the only kind of Orc expression that needs the actual value of the variable —, a variable of undefined value is referenced, Orc suspends the execution of the call until the **where** operator assigns a value to the variable.

As **where** creates only one instance of g, only one publication of f — the first — is used. When f publishes, and since any further processing is vacuous in altering the result of g, it is terminated on the spot and its resources freed.

To the outside world, asynchronous parallel composition relays the publications done by g.

In following example, the family of the previous example will buy a present but only for one of the children — we are in the most dire of times, *n'est-ce pas?* The lucky one will be the one who make his or her choice first.

$$\begin{array}{l}buyfromAmazon(x) \\ \qquad \textbf{where } x :\in (askUser('Son') \mid askUser('Daughter'))\end{array}$$

2.5 Primitive Sites

The core language as presented so far is capable of very little. The designers of Orc address its limitations by adding a few sites to the language with privileged semantics. They provide such goodies as control flow — as one gets in every other language — and timings constructs — fundamental for developing real-time and speculative distributed systems. We refer the reader to [16] for their presentation. In section 3.2 and appendix A we detail our implementation of said sites and the reader can get more information on them there.

2.6 Function Definition

Common sub-expressions may be abstracted through the use of expressions, or as we prefer to call them, functions. Functions are expressions where certain parameters have been abstracted by variables. These variables are introduced in the declaration of the function in a tuple next to its name; this is followed by the expression to which calls to the function reduce.

Functions are called — like sites — on a need to process basis. Only when an orchestration needs to call a function does it expand the definition of the function. With this lazy-like approach it is possible to create recursive definitions without compromising termination of the expansion. The following exemplifying function calls a given site and recursively retries to call it if it doesn't publish before the specified timeout.

$$
\begin{aligned}
&retry(site, timeout) = \\
&\quad isSignal(r) \mathbin{>}b\mathbin{>} ifthenelse(b, retry(site, timeout), let(r)) \\
&\qquad \textbf{where } r :\in (site \mid rtimer(timeout))
\end{aligned}
$$

3 HOrc: Orc as a Haskell Domain Specific Language

HOrc is an implementation of Orc as a domain specific language. It is, thus, embedded within another language — Haskell [10] — and every HOrc program is a syntactically correct program of the host language. The main advantage is obvious: there is no need to write a parser; the Haskell interpreter takes care of it. We also get *for free* a type checker, Haskell's types, classes, values and functions, which can be used without the worry that we'll lose any rigour — Haskell, at least in its pure form, has a well defined semantics and has been used extensively to construct provenly correct programs.

On the other hand, there is some impedance mismatch between the two languages — Orc's syntax is not directly convertible to Haskell's syntax. The necessary adjustments, though, are quite straightforward, almost mechanic. Broadly speaking, site calls and language operators are represented by Haskell functions; variables are denoted by regular Haskell variables; defining a Orc function is done by defining a Haskell function. The conversion "formulas" can be found on table 2.

When we run an orchestration, be it a simple site call or a complex expression

$$\frac{s \text{ is a site call}}{s(arg_1, arg_2, \ldots) \equiv \texttt{s arg1 arg2 ...}} \qquad \frac{f \text{ is a HOrc function}}{f(arg_1, arg_2, \ldots) \equiv \texttt{f arg1 arg2 ...}}$$

$$\frac{f \equiv \texttt{f'}\ g \equiv \texttt{g'}}{f\ >var>\ g \equiv \texttt{f'}\ >>=\backslash\texttt{var ->}\ \ \texttt{g'}} \qquad \frac{f \equiv \texttt{f'}\ g \equiv \texttt{g'}}{f \mid g \equiv \texttt{f' `mplus`}\ \ \texttt{g'}}$$

$$\frac{f \equiv \texttt{f'}\ g \equiv \texttt{g'}}{g \textbf{ where } var :\in f \equiv \texttt{f' `prune`}\ \backslash\texttt{var -> g'}} \qquad \frac{g \equiv \texttt{g'}}{f(var_1, var_2, \ldots) = g \equiv \texttt{f(var1,var2,...)= g'}}$$

Table 2
Conversion between Orc and HOrc

made out of several operators, the basic functioning is the same: compute things and publish them. They all belong to the same type: `Action`. The Haskell definition of said type is

```
data Action a = (Chan (Maybe a) -> IO ())
```

To understand this type, we must first explain how the different components of the orchestration manifest themselves during runtime. In HOrc, every component of the system — site calls and operators — is run on its own thread [13]. When an operator starts a new instance of a sub-expression, it must create a dedicated thread to run the sub-expression.

The different parts of an orchestration interact with each other through the values they publish. These values have to be transported between different threads. Haskell's concurrency library provides a buffered channel type, `Channel a` [11], that can store publications of one thread until the receiving thread can handle them. The pattern of communication is very simple: the values published by a sub-expression are only accessed by its parent expression. The parent expression is responsible for the creation of the channel and for delivering it to the child thread. That is done when the child sub-expression is started by having the associated action parameterized on the channel they share.

Inspecting the type definition of orchestrations, we see that the type of publications is not the same as type of the values written to the channel. This is because there is an additional information that is passed between callee and caller orchestrations: termination. Consider a sequential composition — $f\ >x>\ g$ — and think: when should the thread responsible for the composition terminate? Clearly, only after f and every instance of g terminate, because only then can it be assured that no instance of `g` will publish a value; the same is valid for the other operators. Enclosing the publications in values of type `Maybe` allows us to have both kinds of values in the same channel — publications are represented by `Just x` values while termination is signaled by `Nothing` values.

Since orchestrations are functions, we need to provide them with an argument — the output channel — in order to run them. HOrc provides two functions to this end: `run` and `runC`. The first simply ignores the outputs while the second collects them in a list. For their definition see table 3.

Throughout the remainder of this section we'll focus on how each of the components of HOrc is implemented, dealing both with their syntax and semantics. For simplicity and clarity of the exposition, we will ignore a few small details of the

```
run ::  Action a -> IO ()
run (Act f) = newChan >>= f

runC (Act f) = newChan >>= (\x -> forkIO (f x) >> runC' [] x)
runC' acc x = do  v <- readChan x;
                  case v of
                        Nothing -> return acc;
                        Just p -> runC' (acc ++ [p]) x;
```

Table 3
The functions that set in motion the orchestrations

implementation and a very big one: asynchronous signals [15]. It is possible for the user to cancel the execution of an orchestration by hitting `Ctrl+C`. Haskel then generates an asynchronous signal (`SIGKILL`) that is delivered to the main thread of the orchestration. We use the same mechanism to terminate expressions that publish a value when they are on the right side of a **where**. Handling these signals so that the threads don't enter an inconsistent state is mighty hard, and requires a lot of fine tunning of the order of instructions.

3.1 Sites

Sites perform calculations on their inputs and return the computed value. This can be done locally for small things — e.g. , simple arithmetic instructions — or remotely for more complicated things — e.g. finding matches for a query in the whole internet. Remote calling is possible in Haskell using the IO monad. Fitting simple — i.e. pure — computations into IO values is as easy as adding `return $`. Going for the most general case, we assume all sites publishing values of type `a` must have type `IO a`.

Computing the value and leaving it inside the IO monad is not enough; sites must also send the value to the outside world and signal that they are done. Leaving all these tasks to the programmer would be a) error-prone, b) a bore and c) unnecessary, since it is boilerplate code: call the site, enclose the returned value in a Maybe value, write it to the publication channel, signal termination. So, we provide three `lift*` operators, that given an IO value do all the additional housekeeping needed to create well-behaved orchestrations. Their definitions can found in table 4; in the remainder of this section we will describe them in detail.

Going from the simplest to the most general, we start with `liftIO`. It creates an orchestration out of a computation that never fails, i.e. it always returns a value. For situations where a computation may fail — remote calls are the prime example — and shouldn't publish a value, the function `liftMaybe` is at our service. This function expects the argument IO value to return values inside of type `Maybe`. Errors, which are represented by return values of `Nothing`, are blocked, whereas successful computations, indicated by `Just` values, are published to the outside world. The full semantics of site calls, as prescribed by Orc, is only achieved by `liftMaybe`. We provide `liftIO` because we often don't need the full expressive

```
liftIO ::  IO a -> Action a
liftIO m = Act $ \c -> do{ v <- m;
                           writeChan c (Just v);
                           writeChan c Nothing;
                         }

liftMaybe ::  IO (Maybe a) -> Action a
liftMaybe m = Act $ \c->do{ v<-m;
                            maybe (writeChan c Nothing)
                                  (\v' -> writeChan c (Just v')
                                        >> writeChan c Nothing)
                                  v;
                          }

liftList ::  IO [a] -> Action a
liftList l = Act $ \c->do{ v<-m;
                           foldr (\e u->writeChan c (Just e)>>u)
                                 (writeChan c Nothing)
                                 v;
                         }
```

Table 4
The Haskell definition of the site creators

power of `liftMaybe` and this way we don't need to encapsulate the return value in a `Maybe` value.

The final creator of site calls is a bit of a transvestite. If `liftIO` is more restrictive than `liftMaybe`, this one is more permissive. The semantics of site calls requires that a site publishes at most one value. The `liftList` operator, publishes each of the values that are in the list returned by the argument IO action — *ergo*, possibly more than one. It can be seen as a generalization of `liftMaybe`, where failure is represented by an empty list.

3.2 Primitive Sites

Given our reliance on Haskell threads, the order of execution of instructions is entirely determined by the Haskell run-time system. We cannot, therefore, satisfy Orc's property wherein outstanding primitive sites run before their external counterparts. Instead, we consider them to be like ordinary site calls, the only difference being that they come bundled with the system. To differentiate them from normal Haskell terms, we prefix each of them with `o`. They are very simple applications of the lifting operators to existing functions of the Haskell libraries. We refer the reader to appendix A for their definitions.

Just to show off how easy it is to the language and make use of Haskell's own capabilities, we show here how to define a primitive site that we found very useful when debugging orchestrations. This site simply prints out the value of the argument we pass to it. Normally, in Haskell, this is done using the `print` func-

tion. This functions works for values whose type is an instance of the class `Show` — i.e. Haskell's knows how to represent them with strings. Accordingly, the resulting orchestration should only be applicable to such values. Since this function returns an `IO` value, we simply lift it, and lo and behold, we have ourselves an orchestration `oprint` that makes use of type classes, a feature not found in Orc.

```
oprint::(Show a) => a -> Action ()
oprint = liftIO . print
```

3.3 Sequential Composition and the Action Monad

Consider the sequential composition $f >x> g$. The main task of the operator is to channel the publications of f to instances of g. This is done by replacing the free occurrences of variable x in g by the values published by f. This is similar to the behaviour of β-reduction of the λ-calculus. Using the analogy we can state something like

$$f >x> g(x) \equiv (\lambda x \to g) f$$

The main issue with this statement is that, in Orc, f may publish several values, whereas in lambda calculus any expression reduces, up to normal forms, to one term. What we need is a generalization of β-reduction that applies `g` to every publication of f, i.e.

$$f >x> g \equiv \begin{cases} (\lambda x \to g)\ x_0 \\ (\lambda x \to g)\ x_1 \\ \qquad \cdots \\ (\lambda x \to g)\ x_n \end{cases} \text{ where } x_i :\in f$$

The idea of generalized composition of computations — in our case orchestrations — is formalized by the concept of a monad. A type is a monad whenever it has two operations defined on its values: `return` and bind (>>=). Intuitively, the `return` operation encapsulates a value in a computation that "returns" that value — just like *let*. The bind operation chains computations together, so that the values returned by a computation can be used in latter computations — just like sequential composition. The instantiation of HOrc orchestrations in this class corresponds to HOrc's implementation of sequential composition.

In this section we will provide a detailed explanation of how >>= works under the hood. Because the publications of `f` and of the instances of `g` elicit different responses from the operator, we create two channels to house their publications — one for `f` and one for `g`. The read operation on a channel is blocking. Additionally, there is no `select`-like operation that waits on a group of channels and indicates when one of them has values to be read. This brings us a problem since we don't want to be blocked reading on an empty channel while values are being published to the other.

Clearly, the responsibility for reading the channels and acting accordingly should be split in two distinct threads of execution: one reads the values x_i published by `f` and starts the instances `g(`x_i`)`; the other reads the publications of `g(`x_i`)` and forwards them to the outside world. The latter job we give to the main thread (the one that was started by the execution of the operator), whereas the former is given to an auxiliary thread which we will call spawn.

In HOrc, an operator should only terminate when the threads it initiated have also terminated. Given that now the responsibility of initiating threads is in the spawn thread and not the main thread, there has to be a way to signal latter that there are threads still running.

To that end we use one more channel that counts the number of running child threads. Initially, this channel has one token (for the thread `f`). Whenever `f` publishes, spawn puts one more token in the channel to account for the newly created thread; likewise, when the main thread reads a `Nothing` from the `g`'s output channel, indicating that a thread has finished, it removes a token from the counter. As the thread `f` is also counted in this process, the spawn thread also writes a `Nothing` value in the channel reserved for the instances of `g` when `f` (and itself) terminates. When the counting channel is empty, then all threads are terminated, no new threads can be started nor publications made, so the main thread can also die and retreat to thread-heaven.

3.4 Synchronous Parallel Composition and the Action Monad Plus

The 0 site has interesting connections with sequential and synchronous parallel composition w.r.t. to the publications of an expression; it is the absorbing element of the first, and the neutral element of the latter — if we disregard side-effects resulting from site calls — just like the number 0 is for multiplication and sum, respectively.

Monads that have an absorbing element for sequencing that is also the neutral element of another operation are said to members of the class `MonadPlus`. Haskell has many built-in functions that can be applied to objects which belong to types of this class. To use them we must first instantiate the class `MonadPlus` with our type. That entails defining the 0 object and the corresponding plus operation. For this reason, HOrc represent synchronous parallel composition by `mplus`.

The interaction between the synchronous parallel composition and its child threads is much simpler than that of the sequential composition. In this case, only two threads, one for the left side and another for the right side expression, are under the direct responsibility of the operator's thread. Furthermore, publications of both sub-expression are treated the same, so they can share the same output channel.

Synchronous parallel composition is comprised of two main parts: first comes the initialization of the child threads and the channel to which they publish; then, we read their publications and forward them to the operator's output channel. In this second step we must pay attention to the termination of the child threads. Each of the two sub-expressions must publish a `Nothing`, before the operator's thread can

finish.

To count the number of threads that have still to finish, we abstract the act of processing messages by a function. This function repeatedly reads one publication, forwards it, if applicable, to the calling expression and calls itself recursively to process further messages. One of the arguments of this function is an accumulator that counts the number of threads still running. Whenever it receives a `Nothing` it decrements the value of the accumulator passed to the recursive calls. When the accumulator reaches zero, the function terminates.

3.5 Asynchronous Parallel Composition and Pruning

In [16], the designers of Orc propose two additional different interpretations of the expression g **where** $x :\in f$ than the one they originally laid out. One of them is a simplification wherein the expression g is only evaluated after `f` has published a value and, because it is now irrelevant to the results of the orchestration, has been terminated. This is the one we chose to provide in HOrc. To make clear that our operator does not correspond exactly to Orc's *where*, we have named it `prune` — compared to sequential composition, it prunes some of the branches of the calling tree.

The implementation of `prune` is a simplified version of parallel composition. There is no possibility of concurrent publications of `f` and `g` because `g` only runs after `f` has (voluntarily or not) terminated. As a consequence, we no longer need the auxiliary `spawn` thread, nor the children counter.

Instead, `prune` is composed by two steps. In the first step, `f` and the channel to which it publishes are initialized. `prune` then waits until a value is published or `f` terminates. In the latter case the operator terminates immediately without starting `g` or publishing any value. If `f` does publish a value, then a kill signal is sent to its thread and recursively to its child threads.

In the second step, we initialize a channel to house the publications of `g` and initialize a thread to run `g`. We then proceed to forward all the publications of `g` until it terminates.

4 Implementing Concurrent Programming Patterns with HOrc

In this section we will show only two of the examples we have implemented in HOrc. For the first one we picked up the eight queens problem solution of [16] and re-written it in HOrc to show the differences between the two languages. In the second example we focus in expressing more general programming patterns. We extend the definition of hylomorphism by allowing certain parts to run in parallel. As we later show, with this abstraction it becomes trivial to implement a parallel quicksort algorithm. Among the other programming patterns we implemented in HOrc are MapReduce [7] and workflow patterns [5].

4.1 The Eight Queens Problem

The eight queens problem is a well know combinatorial problem [6] which asks the question: how many ways are there to place eight queens on a chess board, with the restriction that none of them is attacking another, i.e. no two queens should be in the same line, column or diagonal. A most beautiful solution to this problem in Orc, based on backtracking on the solution space, is presented in [16]. Its translation to HOrc is immediate. We show here the HOrc version written in Haskell's `do`-notation, which is simply syntactic sugar for monad sequencing.

```
extend x 1 = msum [check (i:x)| i<-[0..7]]
extend x n = do y <- extend x 1;
                extend y (n-1);
```

In the first line, we make use of Haskell's built-in `msum` function that runs in parallel an instance of `check` for every possible row number `i`. Local site `check` will only publish if adding a queen to board `x` on the `i`th row does not violate the no-attack rule. In the second line, every possible disposition of queens resulting from adding one more queen to the tray is associated with variable `y` and passed, as an accumulator, to the recursive invocation. We can very easily implement `check` as a local site and check the output — all 92 publications — composing `extend [] 8` with `oprint`. The implementation of site `check` can be found in appendix B.

4.2 Parallel Hylomorphisms

Hylomorphisms [3] formalize the idea of divide-and-conquer algorithms. By definition, they are comprised of two separate stages: the anamorphism recursively creates a data structure that represents the call graph of the function; the catamorphism recursively consumes that data structure, computing at each node the output of the hylomorphism. In practice, though, the whole process is optimized by throwing away the intermediate data structure — a process known as *de-forestation* — and combining the so-called *genes* of divide (technically, the anamorphism) and conquer (technically, the catamorphism) processes into a unique recursive stage.

Similarly, the site calls of a system can be chained together, forming a structure much like the call trees from sequential programming. Workflow graphs are a common technique to formalize concurrent applications, indicating how the information flows between the various entities in the system. These can be created programmatically by the anamorphism part of the system, leaving their execution to the catamorphism. The biggest difficulty is to find an appropriate intermediate type that can represent the variety of nodes appearing in the call graph.

The intermediate data structure is defined in terms of products and co-products, corresponding to Cartesian product and disjoint union of types. Therefore, to parallelize the hylomorphism construction, we need first to create parallel operations on both products and co-products, which are easily encoded in HOrc.

4.2.1 Parallel Products

Products represent the accumulation of information. The most important operation is the *split*, which given two functions with equal domain, gathers their results and places them in a tuple. This can be generalized, of course, to other tuple dimensions, and it is also possible to nest binary products to achieve products of higher dimension. What is interesting about this function in the context of parallelization is the fact that the two functions in the split are completely independent and may run simultaneously.

Because we are in the HOrc domain, it is best to rephrase it in its own terms: given two orchestrations that expect an argument of the same type, their split is an orchestration, that receives an input, runs each of the orchestrations in parallel and, when both orchestrations publish *one* value, returns the pair with those publications. Its definition, simply enough, is

```
osplit::(a-> Action b, a-> Action c)-> a-> Action (b,c)
osplit (f,g) a = do p <- newPair;
                    c <-        (f a >>= olet .  Left )
                        `mplus` (g a >>= olet .  Right);
                    putPair p c;
```

The auxiliar type `Pair` buffers the output of each orchestration. It is created by function `newPair`. The individual components of the pair can be written with function `putPair`. The semantics of this function are a bit peculiar: for a `Pair` of type $A \times B$, its input is a value of type $A + B$; if the input is on the left of the co-product it will be written to the first component of the pair while if it is on the right, on the second component. Also, this function is actually a HOrc site which publishes only when it has received values for both components — for this reason `osplit` should only be used with values that publish one and only one value; the value published is the pair with values published by both components. The concrete implementation of this type and its operations can be consulted in appendix C.

A special and very useful kind of split is the product of functions `oprod`. This function is structure preserving: given a pair of functions and a pair of values of the appropriate type, the product of functions applies the first function to the first value in the pair and the second function the second value. Its implementation in HOrc is equal to its definition.

```
oprod::(a-> Action b, c -> Action d)-> (a,c)-> Action (b,d)
oprod (f,g) = osplit (f .  fst, g .  snd)
```

4.2.2 Parallel Co-Products

Co-products, unlike products, cannot be parallelized based solely on their structural components. A co-product $A + B$ represents an alternative: we either have a value of A of a value of B but not both together. Thus, we simply `lift` the existing sequential operation to the HOrc domain.

```
oeither::(a->Action c, b->Action c)->Either a b->Action c
oeither (f,g) = either f g
```

And its structure preserving brethren is

```
osum::(a->Action c,b->Action d)->Either a b->Action (Either c d)
osum (f,g)=oeither(\x->f x>>=olet . Left,\x->g x>>=olet . Right)
```

4.2.3 Parallel Hylomorphisms on Binary Trees

Hylomorphisms come in various forms and shapes — namely those of the intermediate structure. Our hylomorphisms are only parametrized by the gene of the anamorphism and catamorphism, not on the intermediate structure, so we must provide one for each kind. We will focus here on binary trees. Also, we will implement a optimized, de-forested hylomorphism. In this situation, our hylomorphism will in fact be composed by 3 separate stages: the division of tasks made by the gene of the anamorphism; the recursive calls; the catamorphism.

De-forestation also induces us to define our intermediate type in the following way.

```
type Node a b = Either () (a,(b, b))
```

This generalises the type of binary trees, by replacing the recursive part by an additional type variable `b`. The values of type `b` will be fed to the recursive calls. These recursive calls will, in turn, replace those values with their results which will then be consumed by the catamorphism. The definition of our de-forested binary tree hylomorphism is

```
hyloBT::(a->Action(Node b a))->(Node b c->Action c)->a->Action c
hyloBT a c x = a x>>=
        osum(olet,oprod(olet,oprod(hyloBT a c,hyloBT a c)))
              >>= c
```

The parallelization is achieved by the term `oprod (olet,oprod (hyloBT a c,hyloBT a c))`: the recursive calls are independent and the parallel behaviour of `oprod`, inherited from `osplit`, guarantees that they will run in parallel.

As an example, below is a parallel implementation of quicksort [9]. To create it, we picked the ana- and catamorphism genes we knew from the sequential version (cf. chapter 6 of [3]) and brought them to the HOrc domain with the appropriate lifting. [5]

```
oqsort ::  (Ord a) => [a] -> Action [a]
oqsort = hyloBT a c
    where a [] = olet $ Left ();
          a (x:xs) = olet $ Right (x,partition (<x) xs);
          c = oeither(olet . (const []),
                      \(a,(b,c))->olet $! (b++(a:c)))
```

[5] We have also added some strictification to make sure lazy-evaluation doesn't dump the work from one thread to a later stage

As is, all three stages of the hylomorphism are performed sequentially. These stages act as synchronization barriers, hindering parallelization, but they are strictly necessary. We can, however, achieve higher performance by moving around part of the work load of the divide and/or conquer stages. The divide stage splits the input into recursive and non-recursive parts. The non-recursive part, though, is left untouched while the recursive calls are being made. This means it can only processed in the other two stages.

But many times, there is some processing done on the non-recursive part that is independent of the result of the recursive calls and which is suitable to run in parallel with them. A fine example of this are maps that process the non-recursive part, leaving the recursive structure unchanged. An alternative to the above definition of hylomorphisms on binary trees incorporating this idea is

```
hyloBT' ::  (a -> Action (Node b a)) -> (b -> Action c) -> (Node
c d-> Action d) -> a -> Action d
hyloBT' a m c x = a x >>=
          osum(olet,oprod(m,oprod(hyloBT' a m c,hyloBT' a m c)))
                  >>= c
```

The relationship between the variables `a`,`m` and `c` in the definition of `hyloBT'` and variables `a` and `c` in `hyloBT` can be obtained by applying the product-fusion law, and is left as an exercise for the reader.

5 Related Work

This paper should be framed in the context of Orc development, by Misra et al [16,4,5], to which it aims to contribute. Our work, however, differentiates from the "main stream" in many respects. First, we were forced to replace **where** by the less powerful `prune` combinator. So far, though, there has not been a practical example for which we could not rephrase the orchestration definition to use `prune` instead of **where**. The implementation in [4] is based on directed acyclic graph(DAG) traversal; every orchestration is compiled first into a DAG which is then traversed to run the orchestration. All of it is done in Java. Our implementation, on the other hand, is just a Haskell library which lends itself to a seamless integration between HOrc and Haskell: Haskell functions can be run as sites in HOrc and HOrc orchestrations can be used within Haskell programs.

Proença and Clarke attempted earlier, in [18], to re-imagine Orc by representing it in another language, in their case Reo. Our work stands out from theirs in the use of Haskell, a much more popular language, with an extensive library of functions that can be used in orchestrations. Moreover, as shown in this paper, the translation between Orc and HOrc is mechanic. Furthermore, reference [18] claims that Reo cannot handle recursive Orc expressions without being extended; clearly HOrc does not face that problem.

From the opposite side of the spectrum, we can see HOrc as an attempt to introduce distributed/parallel concepts into Haskell. Much work has been done on this field. Concurrent Haskell [11] adds explicit parallelism to the language

by introducing threads and communication primitives among them (Channels and MVars); HOrc is built on top of these primitives. Another approach is the semi-explicit parallelism provided by GpH [14]; instead of mandating a certain division of tasks, the programmer annotates the code, identifying possible division of tasks, and the runtime decides which of those divisions to follow. HOrc seems to lay in between the two approaches: it is higher level than using pure threads but the division of tasks is still fixed.

6 Concluding Remarks

We have presented an implementation of Orc as a functional domain specific language, which allows for quick animation of orchestrations and makes easier the use of this language embedded in real-world applications.

This led, in a natural way, to a distributed encoding of inductive-coinductive recursion patterns, which opens the possibility of scaling up functional program calculi, such as the one introduced in [3], from the *micro*, program-oriented level, to the *macro*, architectural one. We intend to pursue this research path in the future.

Haskell proved to be a most convenient way to experiment with the design of distributed programming languages. The choice to base our implementation in threads suited the needs, but it is not the unique alternative. Our very first approach was based on continuations. It was good enough to express the solution to the 8-queens problem, though woefully inadequate for anything that might block.

A lot of work remains to be done. For example, signal handling between threads is, euphemistically, sub-optimal — while published values are correct, spurious site calls might be made which, in conjunction with side-effects, may lead to unexpected results. Another issue deserving further study concerns the alternative implementation of **where** and its relation with our, simpler but less expressive, `prune` combinator. Finally, work currently being undertaken aims to prove that our implementation is correct by verifying that the semantics of HOrc operators, derived from their Haskell definitions, corresponds to Orc's operators semantics.

References

[1] Arbab, F., *Abstract behaviour types: a foundation model for components and their composition*, in: F. S. de Boer, M. Bonsangue, S. Graf and W.-P. de Roever, editors, *Proc. First International Symposium on Formal Methods for Components and Objects (FMCO'02)*, Springer Lect. Notes Comp. Sci. (2852), 2003 pp. 33–70.

[2] Arbab, F., *Reo: a channel–based coordination model for component composition*, Mathematical Structures in Comp. Sci. **14** (2004), pp. 329–366.

[3] Bird, R. and O. Moor, "The Algebra of Programming," Series in Computer Science, Prentice-Hall International, 1997.

[4] Cook, W. R. and J. Misra, *Implementation outline for orc* (2005).

[5] Cook, W. R., S. Patwardhan and J. Misra, *Workflow patterns in orc*, in: P. Ciancarini and H. Wiklicky, editors, *Coordination Models and Languages, 8th International Conference, COORDINATION 2006, Bologna, Italy, June 14-16, 2006, Proceedings*, Lecture Notes in Computer Science **4038** (2006), pp. 82–96.

[6] Dahl, O.-J., E. W. Dijkstra and C. A. R. Hoare, "Structured Programming," Academic Press (New York NY), 1972.

[7] Dean, J. and S. Ghemawat, *Mapreduce: Simplified data processing on large clusters*, Sixth Symp. on Operating System Design and Implementation (2004), pp. 10–10.

[8] Gelernter, D. and N. Carrier, *Coordination languages and their significance*, Communication of the ACM **2** (1992), pp. 97–107.

[9] Hoare, C. A. R., *Quicksort*, The Computer Journal **5** (1962), pp. 10–15.

[10] Jones, S. P. and et al, editors, "Haskell 98 Language and Libraries, the Revised Report." Cambridge Univ. Press, 2003, 272 pp.

[11] Jones, S. P., A. Gordon and S. Finne, *Concurrent Haskell*, in: *Conference Record of POPL® '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages®*, ACM SIGACT and SIGPLAN (1996), pp. 295–308.

[12] Kitchin, D., W. R. Cook and J. Misra, *A language for task orchestration and its semantic properties*, in: C. Baier and H. Hermanns, editors, *Proc. 17th Inter. Conf. Concurrency Theory, CONCUR 2006, Bonn, Germany, August 27-30* (2006), pp. 477–491.

[13] Launchbury, J. and S. L. P. Jones, *Lazy functional state threads*, in: *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, 1994, pp. 24–35.

[14] Loidl, H.-W., P. Trinder, K. Hammond, S. B. Junaidu, R. G. Morgan and S. L. Peyton Jones, *Engineering parallel symbolic programs in GpH*, Concurrency: Practice and Experience **11** (1999), pp. 701–752.

[15] Marlow, S., S. P. Jones, A. Moran and J. Reppy, *Asynchronous exceptions in haskell*, in: C. Norris and J. J. B. Fenwick, editors, *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, ACM SIGPLAN Notices **36.5** (2001), pp. 274–285.

[16] Misra, J. and W. R. Cook, *Computation orchestration* (2007).

[17] Papadopoulos, G. A. and F. Arbab, *Coordination models and languages*, in: *Advances in Computers* (1998), pp. 329–400.

[18] Proença, J. and D. Clarke, *Coordination models orc and reo compared*, Electr. Notes Theor. Comput. Sci **194** (2008), pp. 57–76.

[19] Rodrigues, N. F. and L. S. Barbosa, *Coordinspector: a tool for extracting coordination data from legacy code*, in: *SCAM '08: Proc. of the Eighth IEEE Inter. Working Conference on Source Code Analysis and Manipulation* (2008), pp. 265–266.

[20] Vos, T. E. J. and D. Swierstra, *Proving distributed hylomorphisms*, UU-CS-2001-40, Informatica Instituut, Utrecht University (2001).

[21] Wedler, C. and C. Lengauer, *On linear list recursion in parallel*, Acta Informatica **35** (1998), pp. 875–909.

[22] Wehrman, I., D. Kitchin, W. R. Cook and J. Misra, *A timed semantics of orc*, Theor. Comput. Sci **402** (2008), pp. 234–248.

A Implementation of Primitive Sites.

Below we show how Orc's primitive sites are defined in HOrc using the `lifting` operators.

The site `ozero` never publishes and terminates ASAP.

```
ozero ::  Action a
ozero = liftMaybe Nothing
```

`olet` simply publishes whatever Haskell value is passed to it, namely tuples.

```
olet ::  a -> Action a
olet x = liftIO $ return x
```

Signals are expressed using Haskell's unit type. Emitting a signal is, therefore, publishing a — the — value of said type

```
osignal ::  Action ()
osignal = olet ()
```

The `oif` site receives values of Haskell `Bool` type and publishes a signal accordingly.

```
oif ::  Bool -> Action ()
oif x = if x then osignal else ozero
```

Sites `oclock`, `ortimer` and `oatimer`, make use of Haskell built-in functions to access the system's clock and thread pausing facilities.

```
oclock ::  Action Int
oclock = liftIO $ do{(TOD s p) <- getClockTime;
                     return (fromInteger s)
                    }

ortimer ::  Action ()
ortimer x = liftIO $ threadDelay (x*10^6)

oatimer ::  Action ()
oatimer x= liftIO $ do{(TOD s p) <- getClockTime;
                       threadDelay (x-(fromInteger s)*10^6)
                      }
```

B Implementation of the `check` site of the 8-queens problem

The site `check` is implemented locally. Function `checkP` is actually the one which takes care of all the logic of the problem, while `check` is its `lifted` version to the

HOrc domain.

```
check = liftMaybe .  return .  checkP

checkP [x] = Just [x]
checkP (x:xs) = if   (all (/=x) xs
                      && diagP x xs 1)
                then Just (x:xs)
                else Nothing

diagP _ [] _ = True
diagP x (y:ys) n =    (x+n) /= y
                   && (x-n) /= y
                   && diagP x ys (n+1)
```

C The Pair ADT implementation

We explained type `Pair` and it's related operations `newPair` and `putPair` in section 4.1. Their definitions are, respectively, the following.

```
type Pair a b = MVar (Maybe a, Maybe b)

newPair ::  Action (Pair a b)
newPair = liftIO $ newMVar (Nothing,Nothing)

putPair ::  Pair a b -> Either a b -> Action (a,b)
putPair p m = liftMaybe $
          do{(a,b) <- takeMVar p;
             (a',b') <- return $ either (\x -> (Just x,b))
                                        (\x -> (a,Just x))
                                         m;
             putMVar p (a',b');
             return (dox<-a';y<- b'; return (x,y));
            }
```