UNIVERSITY OF BIRMINGHAM

University of Birmingham Research at Birmingham

Towards native higher-order remote procedure calls

Fredriksson, Olle; Ghica, Dan; Wheen, Bertram

Link to publication on Research at Birmingham portal

Towards native higher-order remote procedure calls

Olle Fredriksson   Dan R. Ghica   Bertram Wheen

University of Birmingham, UK

Abstract

We present a new abstract machine, called DCESH, which models the execution of higher-order programs running in distributed architectures. DCESH implements a native general remote higher-order function call across node boundaries. It is a modernised version of SECD enriched with specialised communication features required for implementing the remote procedure call mechanism. The key correctness result is that the termination behaviour of the remote procedure call is indistinguishable (bisimilar) to that of a local call. The correctness proofs and the requisite definitions for DCESH and other related abstract machines are formalised using Agda. We also formalise a generic transactional mechanism for transparently handling failure in DCESHs.

We use the DCESH as a target architecture for compiling a conventional call-by-value functional language ("Floskel") which can be annotated with node information. Conventional benchmarks show that the single-node performance of Floskel is comparable to that of OCaml, a semantically similar language, and that distribution overheads are not excessive.

1. Native RPC and transparent distribution

The remote procedure call (RPC) [7] is a widely used mechanism for higher-level inter-process communication. An RPC is what its name suggests: a call to a procedure located on a remote node. The idea is that an RPC looks and acts like an ordinary procedure call, and performs the necessary message passing under the hood. However, the RPC mechanism tends to be bolted on top of a pre-existing language, as a library for example, rather than be seamlessly integrated into it. This leads to significant syntactic differences between calling a local library function and a remote function. Even if these syntactic differences can be smoothed using *stubs* that wrap remote calls into local calls [6] important differences still persist, of which the most important is that arguments must be of ground types.

Generalising RPC to all types and incorporating it seamlessly in the language has been considered but dismissed on grounds of potential inefficiencies [40]. However, considering that a number of technologies that trade efficiency for convenience have been rejected on similar grounds (from machine independent languages to functional programming, garbage collection or automated program verification – to name only a few), we decided it is time to revisit

this issue. Because the execution of applications is increasingly and rapidly moving from single devices to networks of (often heterogeneous) devices a case can be made that such revisiting is timely. A native RPC system, offering as an immediate benefit transparent and automated distribution, can be useful in domains where programmer effectiveness is more important than machine efficiency.

We attack this problem in a principled manner. We want the seamless integration of the RPC in the language to be not merely syntactic but semantic as well: the RPC is a native call of our language, on the same level as a conventional local call. This is realised by introducing a new kind of abstract machine which extends the conventional SECD machine [26], or rather a modernised version of it, with communication primitives. These are not general-purpose low-level communication primitives but are especially designed to support the implementation of RPCs. Even though we aim for simplicity first, some of the technicalities, especially the proofs of correctness, are quite intricate. For this reason the abstract machine network framework and its correctness properties are fully formalised using the Agda language [34]. The technical challenge is not just one of handling complicated formalisms but also mathematical, the key correctness proof requiring the adaptation of the *step-indexed relations* technique [2] to bisimulation.

Note that we are language rather than system oriented. We assume the existence of a run-time infrastructure to handle system-level aspects associated with distribution such as failure detection, load balancing, global reset and initialisation, and so on. From a systems perspective we only deal with failure handling, which is particularly important in a distributed setting, using a general transactional approach.

The networks of abstract machines can serve as a target for the compilation of a conventional call-by-value language which we call Floskel. The interesting new thing about it is that there is almost nothing new about it. It uses a Haskell-like syntax but it has the semantics of the pure fragment of OCaml. Remote procedure calls are indistinguishable from native calls except for an annotation, which can be applied to any sub-term, indicating that it is to be executed on a different node. From the point of view of the language, and of the programmer, this annotation has no syntactic, semantic or typing implication. It is a mere pragma-like directive.

Prior work on native RPCs and seamless distribution is specialised to web programming [11], thus domain-specific. We aim to be as generic as possible in this context. Some existing work uses as a starting point interaction-based semantic paradigms which lend themselves naturally to a communication-centric implementation: Geometry of Interaction [17] and Game Semantics [18]. Such approaches have two significant disadvantages. The exotic operational behaviour makes it impossible to apply known optimisation techniques, and to interact with code compiled conventionally. This can be seen in the low performance of single-node execution of programs compiled using such techniques. On the other hand, the single-node compilation of Floskel is very similar to the conventional compilation of a language such as OCaml, and the bench-

marks indicate that the overhead required by the RPC run-time is not excessive.

In a nutshell, we believe that this paper can help make the case that functional languages with native RPCs could be a lot more useful than presumed.

1.1 Technical outline and contributions

Our main contributions are in the following areas:

Compiler and run-time   We describe the syntax (Sec. 2.1) and implementation (Sec. 2.2) of Floskel, a general-purpose functional language with native RPCs. Our basis is a conventional compiler for such a language, and we show how it is modified to support RPCs and, additionally, *ubiquitous* functions, i.e. functions available on all nodes. Our benchmarks suggest that Floskel's performance is comparable to the state of the art OCaml compiler (Sec. 2.4) for single-node execution.

Abstract machines   The semantics of a core of Floskel has been formalised in Agda (Sec. 3) in the form of an abstract machine that can be used to guide an implementation. To achieve this we make gradual refinements to a machine, based on Landin's SECD machine [26], that we call the *CES machine* (Sec. 3.1). First we add heaps for dynamically allocating closures, forming the *CESH machine* (Sec. 3.2); we show that the CES and CESH execution is bisimilar. We then add communication primitives (synchronous and asynchronous) by defining a general form of networks of nodes that run an instance of an underlying abstract machine (Sec. 3.3). Using these networks, we illustrate the idea of subsuming function calls by communication protocols by constructing a degenerate distributed machine, $DCESH_1$ (Sec. 3.4), that decomposes some machine instructions into message passing, but only runs on one node. Execution on the fully distributed CESH machine called DCESH (Sec. 3.5), is shown to be bisimilar to the CESH machine — our main theoretical result. Finally, we model a general-purpose fault-tolerant environment for DCESH-like machines (Sec. 4) by layering a transactional abstract machine that provides a simple commit-and-rollback mechanism for an abstract machine that may unexpectedly fail.

Formalisation in Agda   The theorems that we present in this paper have been proved correct in Agda [34], an interactive proof assistant and programming language based on intuitionistic type theory. The definitions and proofs in this paper are intricate, so carrying them out manually would be error-prone, arduous and perhaps unconvincing. Agda has been a helpful tool in producing these proofs, providing a pleasant interactive environment in which to play with alternative definitions. To eliminate another source of error, we do not present our results in informal mathematics; the code blocks in Sec. 3 come directly from the formalisation.

The formalisation is organised as follows, where the arrows denote dependence, the lines with $\sim$ symbols bisimulations, and the parenthesised numerals section numbers:
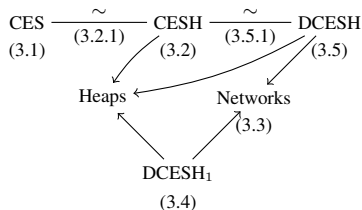
2.   Floskel: a location-aware language

2.1   Syntax

At the core of the Floskel language is a call-by-value functional language with user-definable algebraic data types and pattern matching. Floskel is semantically similar to languages in the ML family, and syntactically similar to Haskell. The main syntactic difference between Floskel and Haskell is that pattern matching clauses are given without the leading function name, to avoid repetition, and that type annotations are given after a single colon, as in the following example:

$$
\begin{aligned}
map &: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
f\,[\,] &= [\,] \\
f\,(x{::}xs) &= f\,x :: map\,f\,xs
\end{aligned}
$$

Node annotations   An ordinary function definition, like *map*, is a *ubiquitous* function by default. This means that it is made available on all nodes in the system, and a call to such a function is always done locally – a plain old function call.

On the other hand, a function or sub-term defined with a *node annotation*, such as

$$
\begin{aligned}
query@Database &: Query \rightarrow Response \\
x &= ...,
\end{aligned}
$$

is *located* and compiled only to the specified node (here *Database*). In the rest of the program *query* can be used like any other function, but the compiler and run-time system treat it differently. A call to *query* from a node other than *Database* is a *remote* call.

Since the programmer can use located functions like any other functions, and this is a functional language, it means that the language has, by necessity, support for higher-order functions across node boundaries. For instance the function

$$
\begin{aligned}
f@A &: (Query \rightarrow Response) \rightarrow X \\
q &= ...\, use\ q\ ...
\end{aligned}
$$

can be applied to *query* yielding $f\,query : X$.

Node annotations can also be applied to sub-expressions, as in the following example:

$$
\begin{aligned}
sum\,[\,] &= 0 \\
(x{::}xs) &= x + sum\,xs \\
xs@A &= ... \\
ys@B &= ... \\
result@C &= (sum\,xs)\,@A + (sum\,ys)\,@B
\end{aligned}
$$

Here we want to calculate the sum, on node *C*, of the elements of two lists located on nodes *A* and *B*. If the lists are lengthy, it is better to calculate the sums on *A* and *B*, and to then send the final sum to *C*, since this saves us having to send the full lists over the network.

2.2   Compilation

The Floskel compiler [1] currently targets C using the MPI library for communication, though other targets are possible since we do not make use of any exotic features of our target. Any compatible combination of low-level language and message passing library would work. Most of the compiler's pipeline is standard for a functional language implementation. It works by applying a series of transformations to the source program until reaching a level low enough to be straightforwardly translated to C. Since the source language has pattern matching, it first compiles the pattern matching to simple case trees [4]. Local definitions are then lifted to the top-level using lambda lifting [25], and lastly the program is closure converted [31] to support partially applied functions.

Up until the lambda lifting, a node annotation is a constructor in the abstract syntax tree of the language's expressions. The lambda lifter lifts such sub-expressions to the top-level such that

annotations are afterwards associated with definitions (and not expressions).

The main work specific to Floskel is done in the closure conversion and the run-time system that the compiled programs make use of.

Closures For function applications, the closure converter distinguishes between known functions – those that are on the top-level and have a known arity, and unknown functions – those that are provided e.g. as function arguments.

A known function f that is either ubiquitous or available on the same node as the definition that is being compiled is compiled to an ordinary function call if there are enough arguments. If there are not, and the function is ubiquitous we have to construct a *partial application* closure, which contains a pointer to the function and the arguments so far. The compiler maintains the invariant that unknown functions are always in the form of a closure, whose general layout is:

g_{ptr}	g_{id}	arity	payload...

Since the function may require access to the payload of the closure, g_{ptr} is a function of arity $arity + 1$: when applying a closure cl as above to arguments $x_1, ..., x_{arity}$, the call becomes $g_{ptr}\ (cl, x_1, ..., x_{arity})$ meaning that the function has access to the payload through cl. To construct the initial closure for a partial application of a function f of arity $arity$ with $nargs$ arguments, we have to conform to this rule, so we construct the closure $(f'_{ptr}, f'_{id}, n, y_1, ..., y_{nargs})$ where $n = arity\ -\ nargs$ and f' is a new ubiquitous top-level function defined as follows:

$f'\ cl\ x_1\ ...\ x_n\ =\ $ case cl of
$\quad (\_, \_, \_, y_1, ..., y_{nargs})\ \rightarrow\ f\ (y_1, ..., y_{nargs}, x_1, ..., x_n)$

A family of $apply_i$ functions handle, in a standard way, applications (of i arguments) of unknown functions by inspecting the arity stored in the closure to decide whether to construct a new partial application closure with the additional arguments or to apply the function.

The field f_{id} is an integer identifier assigned to every function at compile-time and used as a system-wide identifier if the function is ubiquitous, or a node-specific identifier if not. If there are k ubiquitous functions they are assigned the first k identifiers, and the nodes of the system may use identifiers greater than k for their respective located functions. Determining if a function is ubiquitous is thus a simple comparison: $f_{id} < k$. Additionally, every node has a table of functions that maps ubiquitous or local located function identifiers to local function pointers, which is used by the deserialiser.

If we have a saturated call to a known remote function, we make a call to the function $rApply_{arity}$, defined in the run-time system (to be described). If we have a non-saturated call to a known remote or located function, we construct the closure $(f'_{ptr}, f'_{id}, arity, y_1, ..., y_{nargs})$ where f' is a new ubiquitous top-level function defined as follows:

$f'\ cl\ x_1\ ...\ x_n\ =\ $ case cl of
$\quad (\_, \_, \_, y_1, ..., y_{nargs})\ \rightarrow$
$\quad\quad$ if $myNode\ \equiv\ f_{node}$ then
$\quad\quad\quad lookup\ (f_{id})\ (y_1, ..., y_{nargs}, x_1, ..., x_n,)$
$\quad\quad$ else
$\quad\quad\quad rApply_{arity}\ (f_{node}, f_{id}, y_1, ..., y_{nargs}, x_1, ..., x_n)$

Here $myNode$ is the identifier of the node the code is currently being run at. If it is the same node as the node of f, we can make an ordinary function call by looking up the function corresponding to f_{id} in the function table. Otherwise we call the run-time system function $rApply_{arity}$.

In this way, we construct a closure for located functions that looks just like the closure of an ubiquitous function.

2.3 Run-time

The run-time system defines a family of ubiquitous functions $rApply_{arity}$, that, as we saw above, are used for remote procedure calls and to construct closures for located functions. The function takes a function identifier, a node identifier, and *arity* arguments. It serialises the arguments and sends them together with the function identifier to the given node:

$rApply\ f_{node}\ f_{id}\ x_1\ ...\ x_{arity}\ =$
$\quad$ send $(f_{id}, serialise\ (x_1), ..., serialise\ (x_{arity}))$ to f_{node};
$\quad$ receive *answer* from $f_{node}\ \rightarrow$
$\quad$ *answer*

When the node f_{node} receives this message, it looks the function up in its function table, calls it with the deserialised arguments, and sends back the result:

receive $(f_{id}, y_1, ..., y_{arity})$ from *remoteNode* $\rightarrow$
$\quad$ let *result* $=$ *lookup* (f_{id}) $(deserialise\ (y_1), ..., deserialise\ (y_{arity}))$
$\quad$ in send *result* to *remoteNode*

Serialisation In a remote function call the arguments may be values from arbitrary algebraic data types (like lists and trees), in addition to primitive types and functions.

The serialisation of a primitive type is the identity function, while algebraic data-types require a traversal and flattening of the heap structure. We use tags in the lower bits of a value's field to differentiate between pointers and non-pointers, which makes this flattening straightforward. The interesting part of serialisation is how to handle closures, both in the case of ubiquitous and located functions.

For closures around ubiquitous functions, we serialise the closure almost as is, but use the function identifiers to resolve the function pointer on the receiving node, as the pointer is not guaranteed to be the same on each node.

To handle located functions, the most straightforward implementation is to use "mobilised" closures that work by exchanging the located function with a ubiquitous function that calls $rApply$ to perform the remote procedure call – a sort of lambda-lifting for locations. This is what our implementation currently does. Our formalisation will describe an optimised variant of this scheme, which instead saves the closure on the sending node and sends a pointer to that. The optimised scheme means that we do not unnecessarily send closures containing (potentially large) arguments that are going to end up on the node they originated from anyway. The cost of this optimisation, however, is that it requires us to keep track of heap-allocated pointers across node boundaries using distributed garbage collection. The serialisation currently implemented does not require such garbage collection, but may be slow when dealing with large data.

In detail, to serialise a closure

f_{ptr}	f_{id}	arity	payload...

we put a placeholder, CL, in the place of f_{ptr}:

CL	f_{id}	arity	payload' ...

where *payload'* represents the serialised payload and CL is a tag that can be used to identify that this is a closure. To deserialise this on the receiving end, we look up the function pointer associated with f_{id} in the ubiquitous function table and substitute that for CL.

2.4 Performance benchmarks

Single-node Before we measure the performance of the implementation of the native RPC, we analyse how the single-node performance is affected by the distribution overhead even if it is not used — is it feasible for a general-purpose language to be based on the DCESH?

	`trees`	`nqueens`	`qsort`	`primes`	`tak`	`fib`
Floskel	91.2s	12.2s	9.45s	19.3s	16.5s	10.0s
`ocamlopt`	43.0s	3.10s	3.21s	6.67s	2.85s	1.68s
relative	2.12	3.94	2.94	2.9	5.77	5.95

Figure 1. Single-node performance.

	`trees`	`nqueens`	`qsort`	`primes`	`tak`	`fib`
μs/remote call	618	382	4.77	13.4	6.94	6.87
B/remote call	1490	25.8	28.1	27.0	32.0	24.0

Figure 2. Distribution overheads.

Fig. 1 shows absolute and relative timings of a number of small benchmarks using integers, lists, trees, recursion, and a small amount of output for printing results. We compare the performance of Floskel programs compiled with our compiler, and equivalent OCaml programs compiled using `ocamlopt`, a high-performance native-code compiler. Since our compiler targets C, we further compile the generated files to native code using `gcc -O2`. We can see that the running time of programs compiled with our compiler is between two and six times greater than that of those compiled with `ocamlopt`. These results should be viewed in the light of the fact that our compiler only does a minimal amount of optimisation, whereas a considerable amount of time and effort has been put into `ocamlopt`.

Moreover, our compiler only produces C code rather than assembly. Compiling to C rather than assembly, especially in the style we use, prevents the C compiler from using whole-program optimisations and is, therefore, a serious source of inefficiencies.

Distribution overhead We measure the overhead of our implementation of native remote procedure calls by running the same programs as above, but distributed to between two and nine nodes. The distribution is done by adding node annotations in ways that generate large amounts of communication. We run the benchmarks on a single physical computer with local virtual nodes, which means that the contributions of network latencies are factored out. These measurements give the overhead of the other factors related to remote calls, like serialisation and deserialisation. The results are shown in Fig. 2. The first row, μs/remote call, is obtained by running the same benchmark with and without node annotations, taking the delta-time of those two, and then dividing by the number of remote invocations in the distributed program. The second row measures the amount of data transmitted per remote invocation, in bytes.

It is expected that this benchmark depends largely on the kinds of invocations that are done, since it is more costly to serialise and send a long list or a big closure than an integer. The benchmark hints at this; we appear to get a higher cost for remote calls that are big.

An outlier is the `nqueens` benchmark, which does not do remote invocations with large arguments, but still has a high overhead per call, because it intentionally uses many localised functions.

3. Abstract machines and networks

Having introduced the programming language, its compiler and its run-time system we now present the theoretical foundation for the correctness of the compiler. We start with the standard abstract machine model of CBV computation, which we refine, in several steps, into increasingly expressive abstract machines with heap and networking capabilities, while showing along the way that correctness is preserved, via bisimulation results. All definitions and theorems are formalised using the proof assistant Agda, the

syntax of which we will follow. We give some of the key definitions and examples in Agda syntax, but most of the description of the formalisation is intended as a high-level guide to the Agda code, which is available online [1]. In order to help the reader navigate the code when a significant theorem or lemma is mentioned the fully qualified Agda name is given in a footnote. Note that we shall not formalise the whole of Floskel but only a core language which coincides with Plotkin's (untyped) call-by-value PCF [37].

3.1 The CES machine

The starting point is a variation of Landin's SECD machine [26] called Modern SECD [27], which can be traced to the SECD machine variation of Henderson [22] and to the CEK machine of Felleisen [15], which we call CES (Agda module *CES*). Just like the machine of Henderson, it uses bytecode for the control component of the machine, and just like the CEK it places the continuations that originally resided in the dump directly on the stack, simplifying the machine configurations.

A CES configuration (*Config*) is a tuple consisting of a fragment of code (*Code*), an environment (*Env*), and a stack (*Stack*). Evaluation begins with an empty stack and environment, and then follows a stack discipline. Sub-terms push their result on the stack so that their super-terms can consume them. When (and if) the evaluation terminates, the program's result is the sole stack element.

Source language The source language has constructors for lambda abstractions ($\boldsymbol{\lambda}$ *t*), applications (*t* **\$** *t'*), and variables (**var** *n*), represented using De Bruijn indices [13], so a variable is a natural number. Additionally, we have natural number literals (**lit** *n*), binary operations (**op** *f t t'*), and a conditional (**if0** *t* **then** t_0 **else** t_1). Because the language is untyped, we can express a fixed-point combinator without adding additional constructors.

The machine operates on bytecode and does not directly interpret the source terms, so the terms need to be compiled before they can be executed. The main work of compilation is done by the function *compile'*, which takes a term *t* and a fragment of code *c* used as a postlude. The bold upper-case names (**CLOS**, **VAR**, and so on) are the bytecode instructions, which are sequenced using _;_. Instructions can be seen to correspond to the constructs of the source language, sequentialised.

> *compile'* : *Term* → *Code* → *Code*
> *compile'* ($\boldsymbol{\lambda}$ *t*)   *c* = **CLOS** (*compile'* *t* **RET**) **;** *c*
> *compile'* (*t* **\$** *t'*)   *c* = *compile'* *t* (*compile'* *t'* (**APPL ;** *c*))
> *compile'* (**var** *x*)   *c* = **VAR** *x* **;** *c*
> *compile'* (**lit** *n*)   *c* = **LIT** *n* **;** *c*
> *compile'* (**op** *f t t'*) *c* = *compile'* *t'* (*compile'* *t* (**OP** *f* **;** *c*))
> *compile'* (**if0** *b* **then** *t* **else** *f*) *c* =
>   *compile' b* (**COND** (*compile'* *t c*) (*compile' f c*))

Example 3.1. To compile a term *t* we supply **END** as a postlude: *compile t* = *compile'* *t* **END**. The term *t* = ($\lambda x.\ x$) ($\lambda x\ y.\ x$) is compiled as follows:

> *compile* (($\boldsymbol{\lambda}$ **var** *0*) **\$** ($\boldsymbol{\lambda}$ ($\boldsymbol{\lambda}$ **var** *1*))) = **CLOS** (**VAR** *0* **; RET**) **;**
>   **CLOS** (**CLOS** (**VAR** *1* **; RET**) **; RET**) **; APPL ; END**

Environments (*Env*) are lists of values (*List Value*), which are either natural numbers (**nat** *n*) or closures (**clos** *cl*). A closure (*Closure*) is a fragment of code paired with an environment (*Code* × *Env*). Stacks (*Stack*) are lists of stack elements (*List StackElem*), which are either values (**val** *v*) or continuations (**cont** *cl*), represented by closures.

Fig. 3 shows the definition of the transition relation for configurations of the CES machine. A note on Agda syntax: The instruction constructor names are overloaded as constructors for the relation; their usage is disambiguated by context. Arguments in

$$\textbf{VAR} \quad : \forall\,\{n\ c\ e\ s\ v\} \to lookup\ n\ e \equiv \textbf{just}\ v \to \quad (\textbf{VAR}\ n\ ;\ c, e, s)\ \xrightarrow[CES]{}\ (c, e, \textbf{val}\ v :: s)$$

$$\textbf{CLOS} \quad : \forall\,\{c'\ c\ e\ s\} \to \quad (\textbf{CLOS}\ c'\ ;\ c, e, s)\ \xrightarrow[CES]{}\ (c, e, \textbf{val}\ (\textbf{clos}\ (c', e)) :: s)$$

$$\textbf{APPL} \quad : \forall\,\{c\ e\ v\ c'\ e'\ s\} \to \quad (\textbf{APPL}\ ;\ c, e, \textbf{val}\ v :: \textbf{val}\ (\textbf{clos}\ (c', e')) :: s)\ \xrightarrow[CES]{}\ (c', v :: e', \textbf{cont}\ (c, e) :: s)$$

$$\textbf{RET} \quad : \forall\,\{e\ v\ c\ e'\ s\} \to \quad (\textbf{RET}, e, \textbf{val}\ v :: \textbf{cont}\ (c, e') :: s)\ \xrightarrow[CES]{}\ (c, e', \textbf{val}\ v :: s)$$

$$\textbf{LIT} \quad : \forall\,\{n\ c\ e\ s\} \to \quad (\textbf{LIT}\ n\ ;\ c, e, s)\ \xrightarrow[CES]{}\ (c, e, \textbf{val}\ (\textbf{nat}\ n) :: s)$$

$$\textbf{OP} \quad : \forall\,\{f\ c\ e\ n_1\ n_2\ s\} \to (\textbf{OP}\ f\ ;\ c, e, \textbf{val}\ (\textbf{nat}\ n_1) :: \textbf{val}\ (\textbf{nat}\ n_2) :: s)\ \xrightarrow[CES]{}\ (c, e, \textbf{val}\ (\textbf{nat}\ (f\ n_1\ n_2)) :: s)$$

$$\textbf{COND-0} \quad : \forall\,\{c\ c'\ e\ s\} \to \quad (\textbf{COND}\ c\ c', e, \textbf{val}\ (\textbf{nat}\ 0) :: s)\ \xrightarrow[CES]{}\ (c, e, s)$$

$$\textbf{COND-1+n} \quad : \forall\,\{c\ c'\ e\ n\ s\} \to \quad (\textbf{COND}\ c\ c', e, \textbf{val}\ (\textbf{nat}\ (1+\ n)) :: s)\ \xrightarrow[CES]{}\ (c', e, s)$$

Figure 3. The definition of the transition relation of the CES machine.

curly braces are *implicit* and can be automatically inferred. Equality (propositional) is written $\_\equiv\_$.

Stack discipline is clear in the definition of the transition relation. When e.g. **VAR** is executed, the CES machine looks up the value of the variable in the environment and pushes it on the stack. A somewhat subtle part of the relation is the interplay between the **APPL** instruction and the **RET** instruction. When performing an application, two values are required on the stack, one of which has to be a closure. The machine enters the closure, adding the value to the environment, and pushes a return continuation on the stack. The code inside a closure will be terminated by a **RET** instruction, so once the machine has finished executing the closure (and thus produced a value on the stack), that value is returned to the continuation. Note that it will be useful to establish that the CES machine is deterministic.[1]

Example 3.2. We trace the execution of Ex. 3.1 defined above, which exemplifies how returning from an application works. Here we write $a\ \xrightarrow[CES]{}\ \langle\ x\ \rangle\ b$ meaning that the machine uses rule x to transition from a to b.

```
let c₁  =  VAR 0 ; RET
    c₂  =  CLOS (VAR 1 ; RET) ; RET
    cl₁ =  val (clos (c₁, [])); cl₂ = val (clos (c₂, []))
 in (CLOS c₁ ; CLOS c₂ ; APPL ; END, [], [])
 ─────⟨ CLOS ⟩    (CLOS c₂ ; APPL ; END, [], [cl₁])
  CES
 ─────⟨ CLOS ⟩    (APPL ; END, [], [cl₂, cl₁])
  CES
 ─────⟨ APPL ⟩    (VAR 0 ; RET, [cl₂], [cont (END, [])])
  CES
 ─────⟨ VAR refl ⟩ (RET, [cl₂], [cl₂, cont (END, [])])
  CES
 ─────⟨ RET ⟩     (END, [], [cl₂])
  CES
```

The final result is therefore the second closure, cl_2.

The CES machine *terminates with a value* v, written $cfg\ \downarrow_{CES}\ v$ if it, through the reflexive transitive closure of $\xrightarrow[CES]{}$, reaches the end of its code fragment with an empty environment, and v as its sole stack element. It *terminates*, written $cfg\ \downarrow_{CES}$ if there exists a value v such that it terminates with the value v. It *diverges*, written $cfg\ \uparrow_{CES}$ if it is possible to take another step from any configuration reachable from the reflexive transitive closure of $\xrightarrow[CES]{}$.

We do not prove formally that the compilation of CBV-PCF to the CES machine is correct, as it is a standard result [12].

3.2 CESH: A heap machine

In a compiler implementation of the CES machine targeting a low-level language, closures have to be dynamically allocated in a heap. However, the CES machine does not make this dynamic allocation explicit. Now we make it explicit in a new machine, called the

CESH, which is a CES machine with an extra heap component in its configuration.

While heaps are not strictly necessary for a *presentation* of the CES machine, they are of great importance to us. The distributed machine that we will later define needs heaps for persistent storage of data, and the CESH machine forms an intermediate step between that and the CES machine. A CESH configuration is defined as

$$Config\ =\ Code \times Env \times Stack \times Heap\ Closure$$

where *Heap* is a type constructor for heaps parameterised by the type of its content. The only difference in the definition of the configuration constituents, compared to the CES machine, is that a closure value (the **clos** constructor of the *Value* type) does not contain an actual closure, but just a pointer (*Ptr*). The stack is as in the CES machine.

Fig. 4 shows those rules of the CESH machine that are significantly different than the CES: **CLOS** and **APPL**. To build a closure, the CESH allocates it in the heap, using the $\_\blacktriangleright\_$ function, which, given a heap and an element, gives back an updated heap and a pointer to the element. When performing an application, the machine has a *pointer* to a closure, so it looks it up in the heap using the $\_!\_$ function, which, given a heap and a pointer, gives back the element that the pointer points to (if it exists).

A CESH configuration *cfg* can *terminate with a value* v, written as $cfg\ \downarrow_{CESH}\ v$, *terminate* ($cfg\ \downarrow_{CESH}$), or *diverge* ($cfg\ \uparrow_{CESH}$). These are analogous to the definitions for the CES machine, except that the CESH machine is allowed to terminate with *any* heap.

3.2.1 Correctness

To show that our definition of the machine is correct, we construct a bisimulation between the CES and CESH, which given the similarity between the two machines, is almost equality. The difference is dealing with closure values, since the CESH stores pointers rather than closures. The relation for closure values must be parameterised by the heap of the CESH configuration, where the (dereferenced) value of the closure pointer is related to the CES closure.

Formally, the relation is constructed separately for the different components of the machine configurations. For bytecode it is equality, and for closures it is defined component-wise. Values are related only if they have the same head constructor and related constituents: if the two values are number literals, they are related if they are equal; a CES closure and a pointer are related only if the pointer leads to a CESH closure that is in turn related to the CES closure. Environments are related if they have the same list spine and their values are pointwise related. The relation on stacks is defined similarly, using the relation on values and continuations. Finally, two configurations are R_{Cfg}-related if their components are related.

In the formalisation we define heaps and their properties *abstractly*, rather than using a specific heap implementation. The first key property we require is that dereferencing a pointer in a heap

---

[1] *CES.Properties.determinism-CES*

$$\textbf{CLOS} \; : \; \forall \{c' \; c \; e \; s \; h\} \rightarrow \text{let } (h',ptr_{cl}) \; = \; h \blacktriangleright (c',e) \text{ in } (\textbf{CLOS } c' \; ; c,e,s,h) \; \xrightarrow[CESH]{} \; (c,e,\textbf{val } (\textbf{clos } ptr_{cl}) :: s, h')$$

$$\textbf{APPL} \; : \; \forall \{c \; e \; v \; ptr_{cl} \; c' \; e' \; s \; h\} \rightarrow h \; ! \; ptr_{cl} \equiv \textbf{just } (c',e') \rightarrow (\textbf{APPL } ; c,e, \textbf{val } v :: \textbf{val } (\textbf{clos } ptr_{cl}) :: s, h) \; \xrightarrow[CESH]{} \; (c', v :: e', \textbf{cont } (c,e) :: s, h)$$

Figure 4. The definition of the transition relation of the CESH machine (excerpt).

where that pointer was just allocated with a value gives back the same value:

$$\forall \; h \; x \rightarrow \text{let } (h',ptr) \; = \; h \blacktriangleright x \text{ in } h' \; ! \; ptr \equiv \textbf{just } x$$

We will require a preorder $\subseteq$ for *sub-heaps*. The intuitive reading for $h \subseteq h'$ is that h' can be used where h can, i.e. that h' contains at least the allocations of h. The second key property that we require of a heap implementation is that allocation does not overwrite any previously allocated memory cells ($proj_1$ means first projection):

$$\forall \; h \; x \rightarrow h \subseteq proj_1 \; (h \blacktriangleright x)$$

For any two heaps h and h' such that $h \subseteq h'$, if $R_{Cfg} \; cfg \; (c,e,s,h)$, then $R_{Cfg} \; cfg \; (c,e,s,h')$.[2]

Our first correctness result is that R_{Cfg} is a simulation relation.[3] The proof is by cases on the *CES* transition, and, in each case, the *CESH* machine can make analogous transitions. The property mentioned above is then used to show that R_{Cfg} is preserved.

It is helpful to introduce the notion of a *presimulation* relation – a generalisation of a simulation relation that does not require the target states of the transitions to be related – formally defined as:

$$Presimulation \; \_\longrightarrow\_ \; \_\longrightarrow'\_ \; \_R\_ \; =$$
$$\forall \; a \; a' \; b \rightarrow (a \longrightarrow a') \rightarrow a \; R \; b \rightarrow \exists \; \lambda \; b' \rightarrow (b \longrightarrow' b')$$

Then, the inverse of R_{Cfg} is a presimulation.[4] In general, if R is a simulation between relations $\longrightarrow$ and $\longrightarrow'$, R^{-1} is a presimulation, and $\longrightarrow'$ is deterministic at states b related to some a, then R^{-1} is a simulation,[5] from which it follows that R_{Cfg} is a bisimulation, because we have already established that the CESH is deterministic. In particular, if $R_{Cfg} \; cfg_1 \; cfg_2$ then $cfg_1 \; \downarrow_{CES}$ **nat** $n \leftrightarrow cfg_2 \; \downarrow_{CESH}$ **nat** n and $cfg_1 \; \uparrow_{CES} \; \leftrightarrow cfg_2 \; \uparrow_{CESH}$.[6]

To finalise the proof we note that there are configurations in R_{Cfg}. One such example is the initial configuration for a fragment of code: For any c, we have $R_{Cfg} \; (c,[],[]) \; (c,[],[],\emptyset)$ (where $\emptyset$ is the empty heap).

3.3 Network models

In this section we define models for networks with synchronous and asynchronous communication, that are parameterised by an underlying labelled transition system. Both kinds of networks are modelled by two-level transition systems, which is common in operational semantics for concurrent and parallel languages. A *global level* describes the transitions of the system as a whole, and a *local level* the local transitions of the nodes in the system. Synchronous communication is modelled by *rendezvous*, i.e. two nodes have to be ready to send and receive a message at a single point in time. Asynchronous communication is modelled using a "message soup", representing messages currently in transit, that nodes can add and remove messages from, reminiscent of the Chemical Abstract Machine [5].

The model (Agda module *Network*) is parameterised by the underlying transition relation of the machines $\_\vdash\_ \xrightarrow[Machine]{\_} \_$. The sets *Node*, *Machine*, and *Msg* are additional parameters. Elements

of *Node* will act as node identifiers, and we assume that these enjoy decidable equality – in MPI, they would correspond to the so called integer "node ranks". The type *Machine* is the type of the nodes' configurations, and *Msg* the type of messages that the machines can send. The presence of the *Node* argument means that the configuration of a node may know about and can depend on its own identifier. The type constructor *Tagged* is used to separate different kinds of local transitions: A *Tagged Msg* can be τ (i.e. a silent transition), **send** *msg*, or **receive** *msg* (for *msg : Msg*).

A synchronous network (*SyncNetwork*) is an indexed family of machines, *Node* $\rightarrow$ *Machine*, representing the nodes of the system. An asynchronous network (*AsyncNetwork*) is an indexed family of machines together with a list of pending messages (*Node* $\rightarrow$ *Machine*) $\times$ *List Msg*.

Fig. 5 shows the definition of the transition relation for synchronous and asynchronous networks. It uses *update*, a function that corresponds to the usual function update (often written $(f \mid x \mapsto y)$) which updates an element of an indexed family (here relying on the decidable equality of node identifiers).

There are two ways for a synchronous network to make a transition. The first, **silent-step**, occurs when a machine in the network makes a transition tagged with τ, and is allowed at any time. The second, **comm-step**, is the aforementioned rendezvous. A node s first takes a step sending a message, and afterwards a node r (which can be s) takes a step receiving the same message. Asynchronous networks only have one rule, **step**, which can be used if a node steps with a tagged message that "agrees" with the pending messages. If the node *receives* a message, the message has to be in the list *before* the transition. If the node *sends* a message, it has to be there *after*. If the node takes a *silent* step, the list stays the same before and after. This is formalised using a function called *detag*, which creates lists of input and output messages from a tagged message.

Asynchronous networks actually subsume synchronous networks.[7] Going in the other direction is not possible in general, but for some specific instances of the underlying transition relation it is, as we will see later.

3.4 DCESH$_1$: A trivially distributed machine

In higher-order distributed programs containing location specifiers, we will sometimes encounter situations where a function is not available locally. For example, when evaluating the function f in the term $(f \; @ \; A) \; (g \; @ \; B)$, we may need to apply the remotely available function g. Our general idea is to do this by decomposing some instructions into communication. In the example, the function f may send a message requesting the evaluation of g, meaning that the **APPL** instruction is split into a pair of instructions: **APPL-send** and **APPL-receive**.

This section outlines an abstract machine, called DCESH$_1$, which decomposes all application and return instructions into communication. The machine is trivially distributed, because it runs as the sole node in a network, sending messages only to itself. Although it is not used as an intermediate step for the proofs, it is included because it illustrates this decomposition.

A configuration of the DCESH$_1$ machine (*Machine*) is a tuple consisting of a possibly running thread (*Maybe Thread*), a closure heap (*Heap Closure*), and a "continuation heap" (*Heap (Closure $\times$ Stack)*). Since the language is sequential we have at most one thread

---

[2] *CESH.Simulation.HeapUpdate.config*

[3] *CESH.Simulation.simulation*

[4] *CESH.Presimulation.presimulation*

[5] *Relation.presimulation-to-simulation*

[6] *CESH.Bisimulation.termination-agrees*, *CESH.Bisimulation.divergence-agrees*

[7] *Network.*$\longrightarrow$*Sync-to-*$\longrightarrow$*Async*$^+$

running at once. The thread resembles a CES configuration, *Thread* = *Code* × *Env* × *Stack*, but stacks are defined differently. A stack is now a list of values paired with an optional pointer (into the continuation heap), *Stack* = *List Val* × *Maybe ContPtr* (*ContPtr* is a synonym for *Ptr*). When performing an application, when CES would push a continuation on the stack, the DCESH$_1$ machine is going to stop the current thread and send a message, which means that it has to save the continuation and the remainder of the stack in the heap for them to persist the thread's lifetime.

The optional pointer in *Stack* is an element at the *bottom* of the list of values. Comparing it to the definition of the CES machine, where stacks are lists of either values or continuations (just closures), we can picture their relation: Whereas the CES machine stores the values and continuations in a single, contiguous stack, the DCESH$_1$ machine stores first a contiguous block of values until reaching a continuation, at which point it stores (**just**) a pointer to the continuation closure and the rest of the stack.

The definition of closures, values, and environments are otherwise just like in the CESH machine. The machine communicates with itself using two kinds of messages, **APPL** and **RET**, corresponding to the instructions that we are replacing with communication.

Fig. 6 defines the transition relation for the DCESH$_1$ machine, written $m \xrightarrow{tmsg} m'$ for a tagged message *tmsg* and machine configurations m and m'. Most transitions are the same as in the CESH machine, just framed with the additional heaps and the **just** meaning that the thread is running. We elide them for brevity.

The interesting rules are the decomposed application and return rules. When an application is performed, an **APPL** message containing a pointer to the closure to apply, the argument value and a pointer to a return continuation (which is first allocated) is sent, and the thread is stopped (**nothing**). We call such a machine *inactive*. The machine can receive an application message if the thread is not running. When that happens, the closure pointer is dereferenced and entered, adding the received argument to the environment. The stack is left empty apart from the continuation pointer of the received message. When returning from a function application, the machine sends a return message containing the continuation pointer and the value to return.

On the receiving end of that communication, it dereferences the continuation pointer and enters it, putting the result value on top of the stack.

Example 3.3. We trace the execution of Ex. 3.1 in a synchronous network of nodes indexed by the unit type. Heaps with pointer mappings are written $\{ptr \mapsto element\}$. The last list shown in each step is the message list of the asynchronous network.

let h_{cl} = $\{ptr_1 \mapsto (c_1, [])\}$
  h'_{cl} = $\{ptr_1 \mapsto (c_1, []), ptr_2 \mapsto (c_2, [])\}$
  h_{cnt} = $\{ptr_{cnt} \mapsto ((\textbf{END}, []), [], \textbf{nothing})\}$
in (**just** (**CLOS** c_1 ; **CLOS** c_2 ; **APPL** ; **END**, [], [], **nothing**), $\emptyset, \emptyset$), []
$\longrightarrow \langle$ **step CLOS** $\rangle$
(**just** (**CLOS** c_2 ; **APPL** ; **END**, [], [**clos** ptr_1], **nothing**), $h_{cl}, \emptyset$), []
$\longrightarrow \langle$ **step CLOS** $\rangle$

(**just** (**APPL** ; **END**, [], [**clos** ptr_2, **clos** ptr_1], **nothing**), $h'_{cl}, \emptyset$), []
$\longrightarrow \langle$ **step APPL-send** $\rangle$
(**nothing**, h'_{cl}, h_{cnt}), [**APPL** ptr_1 (**clos** ptr_2) ptr_{cnt}]
$\longrightarrow \langle$ **step APPL-receive** $\rangle$
(**just** (**VAR** *0* ; **RET**, [**clos** ptr_2], [], **just** ptr_{cnt}), h'_{cl}, h_{cnt}), []
$\longrightarrow \langle$ **step** (**VAR refl**) $\rangle$
(**just** (**RET**, [**clos** ptr_2], [**clos** ptr_2], **just** ptr_{cnt}), h'_{cl}, h_{cnt}), []
$\longrightarrow \langle$ **step RET-send** $\rangle$
(**nothing**, h'_{cl}, h_{cnt}), [**RET** ptr_{cnt} (**clos** ptr_2)]
$\longrightarrow \langle$ **step RET-receive** $\rangle$
(**just** (**END**, [], [**clos** ptr_2], **nothing**), h'_{cl}, h_{cnt}), []

Comparing this to Example 3.2 we can see that an **APPL-send** followed by an **APPL-receive** amounts to the same thing as the **APPL** rule in the CES machine, and similarly for the **RET** instruction.

3.5 DCESH: The distributed CESH machine

We have so far seen two refinements of the CES machine. We have seen CESH, that adds heaps, and DCESH$_1$, that decomposes instructions into communication in a degenerate network of only one node. Our final refinement is a distributed machine, DCESH, that supports multiple nodes. The main problem that we now face is that there is no centralised heap, but each node has its own local heap. This means that, for supporting higher-order functions across node boundaries, we have to somehow keep references to closures in the heaps of *other* nodes. Another problem is efficiency; we would like a system where we do not pay the higher price of communication for locally running code. The main idea for solving these two problems is to use *remote pointers*, *RPtr* = *Ptr* × *Node*, pointers paired with node identifiers signifying on what node's heap the pointer is located. This solves the heap problem because we always know where a pointer comes from. It can also be used to solve the efficiency problem since we can choose what instructions to run based on whether a pointer is local or remote. If it is local, we run the rules of the CESH machine. If it is remote, we run the decomposed rules of the DCESH$_1$ machine.

The final extension to the term language and bytecode will add support for location specifiers. We add a term construct t @ i, and an instruction **REMOTE** c i for its compilation. The location specifiers, t @ i, are taken to mean that the term t should be evaluated on node i. For compilation, we require that the terms t in all location specification sub-terms t @ i are *closed*. Terms where this does not hold are transformed automatically using lambda lifting [25] (transform every sub-term t @ i to t' = $((\lambda\ fv\ t.\ t)$ @ $i)\ (fv\ t)$). The **REMOTE** c i instruction will be used to start running a code fragment c on node i in the network. We also extend the *compile'* function to handle the new term construct:

$$compile'\ (t\ @\ i)\ c\ =\ \textbf{REMOTE}\ (compile'\ t\ \textbf{RET})\ i\ ;\ c$$

Note that we reuse the **RET** instruction to return from a remote computation.

The definition of closures, values, environments and closure heaps are the same as in the CESH machine, but using *RPtr* instead of *Ptr* for closure pointers.

$$\textbf{APPL-send} \quad : \forall \{c\ e\ v\ ptr_{cl}\ s\ r\ h_{cl}\ h_{cnt}\} \to \text{let } (h'_{cnt}, ptr_{cnt}) = h_{cnt} \blacktriangleright ((c,e),s,r) \text{ in}$$

$$(\textbf{just } (\textbf{APPL} \ ;\ c,e,v :: \textbf{clos } ptr_{cl} :: s, r), h_{cl}, h_{cnt}) \xrightarrow{\textbf{send } (\textbf{APPL } ptr_{cl}\ v\ ptr_{cnt})} (\textbf{nothing}, h_{cl}, h'_{cnt})$$

$$\textbf{APPL-receive} \ : \forall \{h_{cl}\ h_{cnt}\ ptr_{cl}\ v\ ptr_{cnt}\ c\ e\} \to h_{cl}\ !\ ptr_{cl} \equiv \textbf{just } (c,e) \to$$

$$(\textbf{nothing}, h_{cl}, h_{cnt}) \xrightarrow{\textbf{receive } (\textbf{APPL } ptr_{cl}\ v\ ptr_{cnt})} (\textbf{just } (c, v :: e, [\,], \textbf{just } ptr_{cnt}), h_{cl}, h_{cnt})$$

$$\textbf{RET-send} \quad\ : \forall \{e\ v\ ptr_{cnt}\ h_{cl}\ h_{cnt}\} \to$$

$$(\textbf{just } (\textbf{RET}, e, v :: [\,], \textbf{just } ptr_{cnt}), h_{cl}, h_{cnt}) \xrightarrow{\textbf{send } (\textbf{RET } ptr_{cnt}\ v)} (\textbf{nothing}, h_{cl}, h_{cnt})$$

$$\textbf{RET-receive} \quad : \forall \{h_{cl}\ h_{cnt}\ ptr_{cnt}\ v\ c\ e\ s\ r\} \to h_{cnt}\ !\ ptr_{cnt} \equiv \textbf{just } ((c,e),s,r) \to$$

$$(\textbf{nothing}, h_{cl}, h_{cnt}) \xrightarrow{\textbf{receive } (\textbf{RET } ptr_{cnt}\ v)} (\textbf{just } (c, e, v :: s, r), h_{cl}, h_{cnt})$$

Figure 6. The definition of the transition relation of the DCESH_1 machine (excerpt).

The stack combines the functionality of the CES(H) machine, permitting local continuations, with that of the DCESH_1 machine, making it possible for a stack to end with a continuation on another node. A stack element is a value or a (local) continuation signified by the **val** and **cont** constructors. A stack (*Stack*) is a list of stack elements, possibly ending with a (remote) pointer to a continuation, *List StackElem × Maybe ContPtr* (where *ContPtr = RPtr*). Threads and machines are defined like in the DCESH_1 machine. The messages that DCESH can send are those of the DCESH_1 machine but using remote pointers instead of plain pointers, plus a message for starting a remote computation, **REMOTE** $c\ i\ rptr_{cnt}$. Note that sending a **REMOTE** message amounts to sending code in our formalisation, which is something that we would not like to do. However, because no code is generated at run-time, every machine can be "pre-loaded" with all the bytecode it needs, and the message only needs to contain a *reference* to a fragment of code.

Fig. 7 defines the transition relation of the DCESH machine, written $i \vdash m \xrightarrow{tmsg} m'$ for a node identifier i, a tagged message *tmsg* and machine configurations m and m'. The parameter i is taken to be the identifier of the node on which the transition is taking place. For local computations, we have rules analogous to those of the CESH machine, so we omit them and show only those for remote computations. The rules use the function $i \vdash h \blacktriangleright x$ for allocating a pointer to x in a heap h and then constructing a remote pointer tagged with node identifier i from it. When starting a remote computation, the machine allocates a continuation in the heap and sends a message containing the code and continuation pointer to the remote node in question. Afterwards the current thread is stopped.
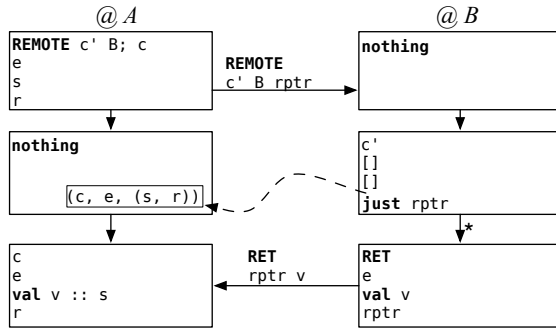
On the receiving end of such a communication, a new thread is started, placing the continuation pointer at the bottom of the stack for the later return to the caller node. To run the apply instruction when the function closure is remote, i.e. its location is *not* equal to the current node, the machine sends a message containing the closure pointer, argument value, and continuation, like in the DCESH_1 machine. On the other end of such a communication, the machine dereferences the pointer and enters the closure with the received value. The bottom remote continuation pointer is set to the received continuation pointer. After either a remote invocation or a remote

application, the machine can return if it has produced a value on the stack and has a remote continuation at the bottom of the stack. To do this, a message containing the continuation pointer and the return value is sent to the location of the continuation pointer. When receiving a return message, the continuation pointer is dereferenced and entered with the received value.

A network of abstract machines is obtained by instantiating the *Network* module with the $\longrightarrow$*Machine* relation. From here on *SyncNetwork* and *AsyncNetwork* and their transition relations refer to the instantiated versions.

Unsurprisingly, if all nodes in a synchronous network except one are inactive, then the next step is deterministic.[8] Another key ancillary property of DCESH networks is that synchronous or asynchronous networks for single threaded computations behave essentially the same,[9] which means it is enough to deal with the simpler synchronous networks.

DCESH network *nodes* can *terminate with a value* v (*nodes* $\downarrow_{Sync}$ v), *terminate* (*nodes* $\downarrow_{Sync}$), or *diverge* (*nodes* $\uparrow_{Sync}$). A network terminates with a value v if it can step to a network where only one node is active, and that node has reached the **END** instruction with the value v on top of its stack. The other definitions are analogous to those of the CES(H) machine.

3.5.1 Correctness

To prove the correctness of the machine, we will now establish a bisimulation between the CESH and the DCESH machines.

To simplify this development, we extend the CESH machine with a dummy rule for the **REMOTE** $c\ i$ instruction so that both machines run the same bytecode. This rule is almost a no-op, but since we are assuming that the code we run remotely is closed, the environment is emptied, and since the compiled code c will end in a **RET** instruction a return continuation is pushed on the stack.

$$(\textbf{REMOTE } c'\ i\ ;\ c, e, s, h) \xrightarrow[CESH]{} (c', [\,], \textbf{cont } (c,e) :: s, h)$$

The relation that we are about to define is, as before, *almost* equality. But since values may be pointers to closures, it must be parameterised by heaps. A technical problem is that *both* machines use pointers, and the DCESH machine also uses *remote* pointers and has two heaps for each node: so the relation must be parameterised by all the heaps in the system. The extra parameter is a synonym for an indexed family of the closure and continuation heaps, *Heaps = Node → DCESH.ClosHeap × DCESH.ContHeap*. The complexity of this relation justifies our use of mechanised reasoning.

The correctness proof itself is not routine. Simply following the recipe that we used before does not work. In the old proof, there can be no circularity, since that bisimulation was constructed inductively on the structure of the CES configuration. But now both systems, CESH and DCESH, have heaps where there is a potential

---

[8] *DCESH.Properties.determinism-Sync*

[9] *DCESH.Properties.$\longrightarrow$Async$^+$-to-$\longrightarrow$Sync$^+$*

$$\begin{aligned}
&\textbf{REMOTE-send} && : \forall \{c'\ i'\ c\ e\ s\ r\ h_{cl}\ h_{cnt}\} \rightarrow \mathsf{let}\ (h'_{cnt}, rptr) = i \vdash h_{cnt} \blacktriangleright ((c,e),s,r)\ \mathsf{in} \\
&&& i \vdash (\textbf{just}\ (\textbf{REMOTE}\ c'\ i'\ ;\ c,e,s,r), h_{cl}, h_{cnt}) \xrightarrow{\textbf{send}\ (\textbf{REMOTE}\ c'\ i'\ rptr)} (\textbf{nothing}, h_{cl}, h'_{cnt}) \\
&\textbf{REMOTE-receive} && : \forall \{h_{cl}\ h_{cnt}\ c\ rptr_{cnt}\} \rightarrow \\
&&& i \vdash (\textbf{nothing}, h_{cl}, h_{cnt}) \xrightarrow{\textbf{receive}\ (\textbf{REMOTE}\ c\ i\ rptr_{cnt})} (\textbf{just}\ (c,[\,],[\,],\textbf{just}\ rptr_{cnt}), h_{cl}, h_{cnt}) \\
&\textbf{APPL-send} && : \forall \{c\ e\ v\ ptr_{cl}\ j\ s\ r\ h_{cl}\ h_{cnt}\} \rightarrow i \not\equiv j \rightarrow \mathsf{let}\ (h'_{cnt}, rptr_{cnt}) = i \vdash h_{cnt} \blacktriangleright ((c,e),s,r)\ \mathsf{in} \\
&&& i \vdash (\textbf{just}\ (\textbf{APPL}\ ;\ c,e,\textbf{val}\ v :: \textbf{val}\ (\textbf{clos}\ (ptr_{cl},j)) :: s, r), h_{cl}, h_{cnt}) \xrightarrow{\textbf{send}\ (\textbf{APPL}\ (ptr_{cl},j)\ v\ rptr_{cnt})} (\textbf{nothing}, h_{cl}, h'_{cnt}) \\
&\textbf{APPL-receive} && : \forall \{h_{cl}\ h_{cnt}\ ptr_{cl}\ v\ rptr_{cnt}\ c\ e\} \rightarrow h_{cl}\ !\ ptr_{cl} \equiv \textbf{just}\ (c,e) \rightarrow \\
&&& i \vdash (\textbf{nothing}, h_{cl}, h_{cnt}) \xrightarrow{\textbf{receive}\ (\textbf{APPL}\ (ptr_{cl},i)\ v\ rptr_{cnt})} (\textbf{just}\ (c, v :: e, [\,], \textbf{just}\ rptr_{cnt}), h_{cl}, h_{cnt}) \\
&\textbf{RET-send} && : \forall \{e\ v\ rptr_{cnt}\ h_{cl}\ h_{cnt}\} \rightarrow \\
&&& i \vdash (\textbf{just}\ (\textbf{RET}, e, \textbf{val}\ v :: [\,], \textbf{just}\ rptr_{cnt}), h_{cl}, h_{cnt}) \xrightarrow{\textbf{send}\ (\textbf{RET}\ rptr_{cnt}\ v)} (\textbf{nothing}, h_{cl}, h_{cnt}) \\
&\textbf{RET-receive} && : \forall \{h_{cl}\ h_{cnt}\ ptr_{cnt}\ v\ c\ e\ s\ r\} \rightarrow h_{cnt}\ !\ ptr_{cnt} \equiv \textbf{just}\ ((c,e),s,r) \rightarrow \\
&&& i \vdash (\textbf{nothing}, h_{cl}, h_{cnt}) \xrightarrow{\textbf{receive}\ (\textbf{RET}\ (ptr_{cnt},i)\ v)} (\textbf{just}\ (c,e,\textbf{val}\ v :: s, r), h_{cl}, h_{cnt})
\end{aligned}$$

Figure 7. The definition of the transition relation of the DCESH machine (excerpt).

for circular references (e.g. a closure, residing in a heap, whose environment contains a pointer to itself), preventing a direct proof via structural induction. This is perhaps the most mathematically (and formally) challenging point of the paper. The solution lies in using the technique of *step-indexed relations*, adapted to the context of bisimulation relations [2]. The additional *rank* parameter records how many times pointers are allowed to be dereferenced.

The rank is used in defining the relation for closure pointers $R_{rptr_{cl}}$. If the rank is zero, the relation is trivially fulfilled. If the rank is non-zero then three conditions must hold. First, the CESH pointer must point to a closure in the CESH heap; second, the remote pointer of the DCESH network must point to a closure in the heap of the location that the pointer refers to; third, the two closures must be related. The relation for stack elements $R_{StackElem}$ is almost as before, but now requires that the relation is true for *any* natural number *rank*, i.e. for any finite number of pointer dereferencings. The relation for stacks R_{Stack} now takes into account that the DCESH stacks may end in a pointer representing a remote continuation, requiring that the pointer points to something in the continuation heap of the location of the pointer, which is related to the CESH stack element. Finally, a CESH configuration and a DCESH thread are R_{Thread}-related if the thread is running and the constituents are pointwise related. Then a CESH configuration is related to a synchronous network R_{Sync} if the network has exactly one running machine that is related to the configuration.

DCESH network heaps are ordered pointwise (called $\subseteq_s$ since it is the "plural" of $\subseteq$). For any CESH closure heaps h and h' such that $h \subseteq h'$ and families of DCESH heaps hs and hs' such that $hs \subseteq_s hs'$, if $R_{Env}\ n\ h\ hs\ e_1\ e_2$ then $R_{Env}\ n\ h'\ hs'\ e_1\ e_2$ and if $R_{Stack}\ h\ hs\ s_1\ s_2$ then $R_{Stack}\ h'\ hs'\ s_1\ s_2$.[10]

Showing that R_{Sync} is a simulation relation[11] proceeds by cases on the CESH transition. In each case, the DCESH network can make analogous transitions. The property above is then used to show that R_{Sync} is preserved. It is quite immediate that the inverse of R_{Sync} is a presimulation[12] which leads to the main result that R_{Sync} is a bisimulation.[13]

In particular, if $R_{Sync}\ cfg\ nodes$ then $cfg \downarrow_{CESH} \textbf{nat}\ n \leftrightarrow nodes \downarrow_{Sync} \textbf{nat}\ n$ and $cfg \uparrow_{CESH} \leftrightarrow nodes \uparrow_{Sync}$,[14] and we also have that initial

configurations are in R_{Sync}.[15] These final results complete the picture for the DCESH machine. We have established that we get the same final result regardless of whether we choose to run a fragment of code using the CES, the CESH, or the DCESH machine.

4. Fault-tolerance via transactions

In this section we present a generic transaction-based method for handling failure which is suitable for the DCESH. Node state is "backed up" (*commit*) at certain points in the execution, and if an exceptional condition arises, the backup is restored (*roll-back*).

This development is independent of the underlying transition relation, but the proofs rely on sequentiality. We assume that we have two arbitrary types *Machine* and *Msg*, as well as a transition relation over them:

$$\_ \xrightarrow[Machine]{\_} \_ : Machine \rightarrow Tagged\ Msg \rightarrow Machine \rightarrow \star$$

Since we have no knowledge of exceptional states in *Machine*, since it is a parameter, we define another relation, $\_ \xrightarrow[Crash]{\_} \_$, as a thin layer on top of $\_ \xrightarrow[Machine]{\_} \_$. The new definition is shown in Fig. 8 and adds the exceptional state **nothing** by extending the set of states of the relation to *Maybe Machine*. The fallible machine can make a **normal-step** transition from and to **just** ordinary *Machine* states, or it can **crash** which leaves it in the exceptional state. This means that we tolerate fail-stop faults as opposed to e.g. the more general Byzantine failures.

The additional assumptions for sequentiality are that we have a decidable predicate, *active* : *Machine* $\rightarrow \star$ on machines, and the following functions:

$$\begin{aligned}
&\textit{inactive-receive-active} : \forall \{m\ m'\ msg\} \rightarrow \\
&\quad (m \xrightarrow[Machine]{\textbf{receive}\ msg} m') \rightarrow \neg\ (active\ m) \times active\ m' \\
&\textit{active-silent-active} \quad : \forall \{m\ m'\} \rightarrow
\end{aligned}$$

---

[10] *DCESH.Simulation-CESH.HeapUpdate.env*, *DCESH.Simulation-CESH.HeapUpdate.stack*

[11] *DCESH.Simulation-CESH.simulation-sync*

[12] *DCESH.Simulation-CESH.presimulation-sync*

[13] *DCESH.Simulation-CESH.bisimulation-sync*

[14] *DCESH.Simulation-CESH.termination-agrees-sync*, *DCESH.Simulation-CESH.divergence-agrees-sync*

[15] *DCESH.Simulation-CESH.initial-related-sync*

$$\begin{aligned}
&\textbf{normal-step} : \forall \{tmsg\ m\ m'\} \rightarrow \\
&\quad (m \xrightarrow[Machine]{tmsg} m') \rightarrow (\textbf{just}\ m \xrightarrow[Crash]{tmsg} \textbf{just}\ m') \\
&\textbf{crash} : \forall \{m\} \rightarrow \\
&\quad (\textbf{just}\ m \xrightarrow[Crash]{\tau} \textbf{nothing})
\end{aligned}$$

Figure 8. The definition of the transition relation of a machine that may crash.

$$\textbf{silent-step} \; : \; \forall \, \{m \; n \; m'\} \rightarrow$$
$$(\textbf{just} \; m \; \xrightarrow[Crash]{\tau} \; \textbf{just} \; m') \quad \rightarrow ((m, n) \; \xrightarrow[Backup]{\tau} \; (m', n))$$
$$\textbf{receive-step} \; : \; \forall \, \{m \; n \; m' \; msg\} \rightarrow$$
$$(\textbf{just} \; m \; \xrightarrow[Crash]{\textbf{receive} \; msg} \; \textbf{just} \; m') \rightarrow ((m, n) \; \xrightarrow[Backup]{\textbf{receive} \; msg} \; (m', m'))$$
$$\textbf{send-step} \; : \; \forall \, \{m \; n \; m' \; msg\} \rightarrow$$
$$(\textbf{just} \; m \; \xrightarrow[Crash]{\textbf{send} \; msg} \; \textbf{just} \; m') \quad \rightarrow ((m, n) \; \xrightarrow[Backup]{\textbf{send} \; msg} \; (m', m'))$$
$$\textbf{recover} \; : \; \forall \, \{m \; n\} \rightarrow$$
$$(\textbf{just} \; m \; \xrightarrow[Crash]{\tau} \; \textbf{nothing}) \quad \rightarrow ((m, n) \; \xrightarrow[Backup]{\tau} \; (n, n))$$

Figure 9. The definition of the transition relation for a crashing machine with backup.

$$(m \; \xrightarrow[Machine]{\tau} \; m') \rightarrow active \; m \times active \; m'$$
$$active\text{-}send\text{-}inactive \quad : \; \forall \, \{m \; m' \; msg\} \rightarrow$$
$$(m \; \xrightarrow[Machine]{\textbf{send} \; msg} \; m') \rightarrow active \; m \times \neg \, (active \; m')$$

These functions express the property that if a machine is invoked, i.e. it receives a message, then it must go from an inactive to an active state. If the machine then takes a silent step, it must remain active, and when it sends a message it must go back to being inactive. This gives us sequentiality; a machine cannot fork new threads, and cannot be invoked several times in parallel.

As the focus here is on obvious correctness and simplicity, we abstract from the method of detecting faults in a separate node, and assume that it can be done (using e.g. a heartbeat network). Similarly, we assume that we have a means of creating and restoring a backup of a node in the system; how this is done depends largely on the underlying system. We so define a machine with a backup as *Backup = Machine × Machine*, where the second *Machine* denotes the backup. Using this definition, we define a backup strategy, given in Fig. 9. This strategy makes a backup just after sending and receiving messages. In the case of the underlying machine crashing, it restores the backup. Note that this is only one of many possible backup strategies. This one is particularly nice from a correctness point-of-view, because it makes a backup after every observable event, although it may not be the most performant.

We define binary relations for making transition with *some* tagged message, as follows:

$$\_ \; \xrightarrow[Machine]{} \; \_ \; : \; Machine \rightarrow Machine \rightarrow \star$$
$$m_1 \; \xrightarrow[Machine]{} \; m_2 \; = \; \exists \, \lambda \; tmsg \rightarrow (m_1 \; \xrightarrow[Machine]{tmsg} \; m_2)$$
$$\_ \; \xrightarrow[Backup]{} \; \_ \; : \; Backup \rightarrow Backup \rightarrow \star$$
$$b_1 \; \xrightarrow[Backup]{} \; b_2 \; = \; \exists \, \lambda \; tmsg \rightarrow (b_1 \; \xrightarrow[Backup]{tmsg} \; b_2)$$

Using these relations we can define the observable trace of run of a *Machine* (*Backup*), i.e. an element of the reflexive transitive closure of the above relations. First we define *IO*, the subset of tagged messages that we can observe, namely **send** and **receive**:

```
data IO (A : ⋆) : ⋆ where
    send receive : A → IO A
```

The following function now gives us the observable trace, given an element of $\xrightarrow[Machine]{}^*$ (which is defined using list-like notation) by ignoring any silent steps.

$$[\![\_]\!]_M \; : \; \forall \, \{m_1 \; m_2\} \rightarrow m_1 \; \xrightarrow[Machine]{}^* \; m_2 \rightarrow List \, (IO \; Msg)$$
$$[\![\, [\,] \,]\!]_M \; = \; [\,]$$
$$[\![\, ((\tau \quad , \_) :: steps) \,]\!]_M \; = \; [\![steps]\!]_M$$
$$[\![\, ((\textbf{send} \quad msg, \_) :: steps) \,]\!]_M \; = \; \textbf{send} \quad msg :: [\![steps]\!]_M$$
$$[\![\, ((\textbf{receive} \; msg, \_) :: steps) \,]\!]_M \; = \; \textbf{receive} \; msg :: [\![steps]\!]_M$$

$[\![\_]\!]_B$ is defined analogously. Given this definition, we can trivially prove that if we have a run $m_1 \; \xrightarrow[Machine]{}^* \; m_2$ then there exists a run of the *Backup* machine that starts and ends in the same state and has the same observational behaviour.[16] This is proved by constructing a crash-free *Backup* run given the *Machine* run. Obviously, the interesting question is whether we can take any *crashing* run and get a corresponding *Machine* run.

The key to proving the result that we want, which more formally is that, given $(bs \; : \; (b_1, b_1) \; \xrightarrow[Backup]{}^* \; (m_2, b_2))$, there is a run $(ms \; : \; b_1 \; \xrightarrow[Machine]{}^* \; m_2)$ with the same observational behaviour as bs[17], is the following lemma[18]:

$$fast\text{-}forward\text{-}to\text{-}crash \; : \; \forall \, \{m_1 \; m_2 \; b_1 \; b_2 \; n\} \rightarrow$$
$$(s \; : \; (m_1, b_1) \; \xrightarrow[Backup]{}^* \; (m_2, b_2)) \rightarrow$$
$$thread\text{-}crashes \; s \rightarrow length \; s \leqslant n \rightarrow$$
$$\exists \, \lambda \, (s' \; : \; ((b_1, b_1) \; \xrightarrow[Backup]{}^* \; (m_2, b_2))) \rightarrow$$
$$(\neg \; thread\text{-}crashes \; s') \times ([\![s]\!]_B \equiv [\![s']\!]_B) \times (length \; s' \leqslant n)$$

Here *thread-crashes* is a decidable property on backup runs, that ensures that, if m_1 is active, then it crashes and does a recovery step at some point before it performs an observable action. The proof of *fast-forward-to-crash* is done by induction on the natural number *n*.

The above result can be enhanced further by observing that if the probability of a machine crash is not 1 then the probability of the machine *eventually* having a successful execution is 1. This means that the probability for the number *n* above to exist is also 1. This argument has not been formalised in Agda.

5. Related work

There is a multitude of programming languages and libraries for distributed or client server computing. We focus mostly on those with a functional flavour. For surveys, see [28, 41]. Broadly speaking, we can divide them into those that use some form of explicit message passing, and those that have more implicit mechanisms for distribution and communication.

Explicit A prime example of a language for distributed computing that uses explicit message passing is Erlang [3]. Erlang is a very successful language used prominently in the telecommunication industry. Conceptually similar solutions include MPI [21] and Cloud Haskell [14]. The theoretically advanced projects Nomadic Pict [43] and the distributed join calculus [16] both support a notion of mobility for distributed agents, which enables more expressivity for the distribution of a program than the fairly static networks (with only ubiquitous *functions* being mobile) that our work uses. Session types have been used to extend a variety of languages, including functional languages, with better communication primitives [42]. Work on session types has been an inspiration also for us: the way that we compile a single program to multiple nodes can be likened to the projection operator in multiparty session types [24]. But in general, explicit languages are well-proven, but far away in the language design-space from the seamless distributed computing that we envision could be done using native RPC, because they place the significant burden of explicit communication on the programmer.

Implicit Our work generalises remote procedure call (RPC) [7] with full support for higher-order functions. In *loc. cit.* it is argued that emulating a shared address space is infeasible since it requires each pointer to also contain location information, and that it is

---

[16] *Backup.soundness*

[17] *Backup.completeness*

[18] *Backup.fast-forward-to-crash*

questionable whether acceptable efficiency can be achieved. These arguments certainly apply to our work, where we do just this. With the goal of expressivity in mind, however, we believe that we should *enable* the programmer to write the potentially inefficient programs that (internally) use remote pointers, because not all programs are performance critical. Furthermore, using a tagged pointer representation [30] for closure pointers means that we can tag pointers that are remote, and pay a very low, if any, performance penalty for local pointers.

Remote Evaluation (REV) [39] is another generalisation of RPC, siding with us on enabling the use of higher-order functions across node boundaries. The main differences between REV and our work is that REV relies on sending code, whereas we can do both, and that it has a more elaborate distribution mechanism.

The well-researched project Eden [29] and the associated abstract machine DREAM [8], which builds on Haskell, is a semi-implicit language. Eden allows expressing distributed algorithms at a high level of abstraction, and is mostly implicit about communication, but explicit about process creation. Eden is specified operationally using a two-level semantics similar to ours, but in the context of the call-by-need evaluation strategy. Kanor [23] is a project that similarly aims to simplify the development of distributed programs by providing a declarative language for specifying communication patterns inside an imperative host language.

Hop [38], Links [11], and ML5 [32] are examples of so called *tierless* languages that allow writing (for instance) the client and server code of web applications in unified languages with more or less seamless interoperability between them. The Links language shares our goal of unifying distributed programs into a single language with seamless interoperability between the nodes, but its focus is on web programming with client, server, and database. For that purpose it includes sub-languages for elegantly constructing and manipulating XML documents and for doing database queries. The behaviour of Links is specified using the client/server calculus [10], which is an operational semantics similar in purpose to our abstract machines, but, as its name suggests, limited to two nodes. The two nodes of the system are constructed with web-programming in mind and are not equal peers. The server is stateless to be able to handle a large number of clients and unexpected disconnections. The semantics operates on a first-order language and uses explicit substitutions. The client/server calculus also inspired work on the LSAM [33], that lifts the two-node limitation. A similar, but earlier, machine is that of dML [35]. The dML and LSAM machines are conceptually close to our machine, but described on a level that is not as readily implementable as our work, using explicit substitutions and synchronous message passing.

On the language side, our work draws inspiration from abstract machines for game semantics [17, 18] where an exotic compilation technique based on game semantics is used to implement a language like ours, but for call-by-name and without algebraic data types. Recent work shows a formalisation similar to ours, but based on the Krivine machine and the significantly less popular call-by-name evaluation strategy [19].

6.   Conclusion

The main conceptual contribution of this work is a new abstract machine, or abstract machine network rather, called the DCESH. Its main feature is that function calls behave, from the point of view of the programmer, in the same way whether they are local or remote. Moreover, on a single node the behaviour of DCESH is that of a conventional SECD-like abstract machine. All correctness proofs have been formalised using Agda.

On this theoretical foundation it is natural to build a compiler for a (very conventional) CBV language where terms are location-aware. Compared to most of the literature on the topic, the thrust of this work is not *a general-purpose functional language for location-aware computing* but rather *a location-aware compilation technique for general-purpose functional languages.* The performance of the compiler, as suggested by the benchmarks, improves dramatically on previous work in terms of single-node behaviour, and the distribution overheads are not onerous.

RPC as a paradigm has been criticised for several reasons: its execution model does not match the native call model, there is no good way of dealing with failure, and it is inherently inefficient [40]. By taking an abstract machine model in which RPCs behave exactly the same as local calls, by showing how a generic transaction mechanism can handle failure, and by implementing a reasonably performant compiler we address all these problems head-on. We believe that we provide enough evidence for general native RPCs to be reconsidered in a more positive light.

6.1   Further work

The DCESH has the internal machinery required for parallel execution, but we restrict ourselves to sequential execution. In moving towards parallelism there are several language design (how to add parallelism?) and theoretical (is compilation still correct?) challenges. The language design aspects are too broad to discuss here, beyond emphasising that the thread-based mechanism of DCESH is indeed quite flexible. The only ingredient lacking is a synchronisation primitive, but that is not a serious difficulty. The theoretical challenges are mainly stemming out of the failure of the equivalence of synchronous and asynchronous networks in the presence of multiple pending messages.[19] Whether we choose to stick to a synchronous network model, which can however be rather unrealistic, or we try to work directly in the more challenging environment of asynchronous networks, remains to be seen.

The current implementation does not need distributed garbage collection. The values which are the result of CBV evaluation are always sent, along with any required closures, to the node where the function using them as arguments is executed. With this approach local garbage collection suffices. Note that this is similar to the approach that Links takes. If a large data structure needs to be held on a particular node the programmer needs to be aware of this requirement and indirect the access to it using functions. However, if we wanted to automate this process as well, and prevent some data from migrating when it's too large, the current approach to garbage collection could not cope, and distributed garbage collection would be required. Mutable references or lazy evaluation would also require it. Whether this can be done efficiently is a separate topic of research [36].

Whether code or data can or should be migrated to different nodes is a question that can be answered from a safety or from an efficiency point of view. The safety angle is very well covered by type systems such as ML5's, which prevent the unwanted exportation of local resources. The efficiency point of view can also be dealt with in a type-theoretic way, as witnessed by recent work in resource-sensitive type systems [9, 20]. The flexibility of the DCESH in terms of localising or remoting the calls (statically or even dynamically) together with a resource oriented system can pave the way towards a highly-convenient automatic orchestration system in which a program is automatically distributed among several nodes to achieve certain performance objectives.

Finally, an Agda formalisation is given only for the abstract machine and its properties, which are the new theoretical contributions of the paper. However, a full formalisation of the compiler stack remains a long-term ambition.

---

[19] $DCESH.Properties. \longrightarrow Async^+ \text{-}to\text{-} \longrightarrow Sync^+$

References

[1] Floskel, proofs and compiler implementation. `http://www.cs.bham.ac.uk/~ohf162/floskel.tar.gz`. Last accessed: 3 June 2014.

[2] A. W. Appel and D. A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS*, 23(5):657–683, 2001. . URL `http://doi.acm.org/10.1145/504709.504712`.

[3] J. Armstrong, R. Virding, and M. Williams. *Concurrent programming in ERLANG*. Prentice Hall, 1993. ISBN 978-0-13-285792-5.

[4] L. Augustsson. Compiling pattern matching. In *FPCA*, pages 368–381, 1985.

[5] G. Berry and G. Boudol. The Chemical Abstract Machine. In *POPL*, pages 81–94, 1990.

[6] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Trans. Comput. Syst.*, 8(1):37–55, 1990.

[7] A. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[8] S. Breitinger, U. Klusik, R. Loogen, Y. Ortega-Mallén, and R. Pena. Dream: The distributed eden abstract machine. In *IFL*, pages 250–269, 1997.

[9] A. Brunel, M. Gaboardi, D. Mazza, and S. Zdancewic. A core quantitative coeffect calculus. In *ESOP*, pages 351–370, 2014.

[10] E. Cooper and P. Wadler. The RPC calculus. In *PPDP*, pages 231–242, 2009.

[11] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web Programming Without Tiers. In *FMCO*, pages 266–296, 2006.

[12] O. Danvy and K. Millikin. A rational deconstruction of Landin's SECD machine with the J operator. *LMCS*, 4(4), 2008.

[13] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, pages 381–392, 1972. .

[14] J. Epstein, A. P. Black, and S. L. P. Jones. Towards Haskell in the cloud. In *Symposium on Haskell 2011*, pages 118–129.

[15] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine, and the lambda-calculus. In *IFIP TC 2/WG 2.2*, Aug. 1986.

[16] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *CONCUR*, pages 406–421, 1996.

[17] O. Fredriksson and D. R. Ghica. Seamless Distributed Computing from the Geometry of Interaction. In *TGC*, pages 34–48, 2012.

[18] O. Fredriksson and D. R. Ghica. Abstract Machines for Game Semantics, Revisited. In *LICS*, pages 560–569, 2013.

[19] O. Fredriksson and D. R. Ghica. Krivine Nets: A semantic foundation for distributed execution. In *ICFP*, 2014 (to appear).

[20] D. R. Ghica and A. I. Smith. Bounded linear types in a resource semiring. In *ESOP*, pages 331–350, 2014.

[21] W. D. Gropp, E. L. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT Press, 1999.

[22] P. Henderson. *Functional programming - application and implementation*. Prentice Hall International Series in Computer Science. 1980. ISBN 978-0-13-331579-0.

[23] E. Holk, W. E. Byrd, J. Willcock, T. Hoefler, A. Chauhan, and A. Lumsdaine. Kanor - a declarative language for explicit communication. In *PADL*, pages 190–204, 2011.

[24] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284, 2008.

[25] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *FPCA*, pages 190–203, 1985.

[26] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, Jan. 1964.

[27] X. Leroy. MPRI course 2-4-2, part II: abstract machines. 2013-2014. URL `http://gallium.inria.fr/~xleroy/mpri/progfunc/`.

[28] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. Michaelson, R. Pena, S. Priebe, Á. J. R. Portillo, and P. W. Trinder. Comparing Parallel Functional Languages: Programming and Performance. *HOSC*, 16(3):203–251, 2003.

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

[30] S. Marlow, A. R. Yakushev, and S. L. P. Jones. Faster laziness using dynamic pointer tagging. In *ICFP*, pages 277–288, 2007.

[31] Y. Minamide, J. G. Morrisett, and R. Harper. Typed closure conversion. In *POPL*, pages 271–283, 1996.

[32] T. Murphy VII, K. Crary, and R. Harper. Type-Safe Distributed Programming with ML5. In *TGC 2007*, pages 108–123.

[33] K. Narita and S.-y. Nishizaki. A parallel abstract machine for the RPC calculus. In *Informatics Engineering and Information Science*, pages 320–332. Springer, 2011.

[34] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers Uni. of Tech., 2007.

[35] A. Ohori and K. Kato. Semantics for communication primitives in an polymorphic language. In *POPL*, pages 99–112, 1993.

[36] D. Plainfossé and M. Shapiro. A Survey of Distributed Garbage Collection Techniques. In *IWMM*, pages 211–249, 1995.

[37] G. D. Plotkin. LCF Considered as a Programming Language. *Theor. Comput. Sci.*, 5(3):223–255, 1977.

[38] M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the web 2.0. In *OOPSLA*, pages 975–985, 2006.

[39] J. W. Stamos and D. K. Gifford. Remote evaluation. *TOPLAS*, 12(4):537–565, 1990.

[40] A. S. Tanenbaum and R. van Renesse. *A critique of the remote procedure call paradigm*. Vrije Universiteit, Subfaculteit Wiskunde en Informatica, 1987.

[41] P. W. Trinder, H.-W. Loidl, and R. F. Pointon. Parallel and Distributed Haskells. *JFP*, 12(4&5):469–510, 2002.

[42] V. T. Vasconcelos, S. J. Gay, and A. Ravara. Type checking a multi-threaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006.

[43] P. T. Wojciechowski and P. Sewell. Nomadic pict: language and infrastructure design for mobile agents. *IEEE Concurrency*, 8(2):42–52, 2000.