# Krivine Nets

## A semantic foundation for distributed execution

Olle Fredriksson Dan R. Ghica
University of Birmingham

## Abstract

We define a new approach to compilation to distributed architectures based on networks of abstract machines. Using it we can implement a generalised and fully transparent form of Remote Procedure Call that supports calling higher-order functions across node boundaries, without sending actual code. Our starting point is the classic Krivine machine, which implements reduction for untyped call-by-name PCF. We successively add the features that we need for distributed execution and show the correctness of each addition. Then we construct a two-level operational semantics, where the high level is a network of communicating machines, and the low level is given by local machine transitions. Using these networks, we arrive at our final system, the *Krivine Net*. We show that Krivine Nets give a correct distributed implementation of the Krivine machine, which preserves both termination and non-termination properties. All the technical results have been formalised and proved correct in AGDA. We also implement a prototype compiler which we compare with previous distributing compilers based on Girard's Geometry of Interaction and on Game Semantics.

***Categories and Subject Descriptors*** D.3.1 [*Programming Languages*]: Formal Definitions and Theory—semantics; D.3.2 [*Programming Languages*]: Language Classifications—concurrent, distributed, and parallel languages; F.1.1 [*Computation by Abstract Devices*]: models of computation

***Keywords*** abstract machines; distributed execution; simulation relation; Agda

## 1. Seamless distribution

There are two extreme views of programming languages. At one extreme we have the *machine-oriented* view, where the programming language is construed as the medium through which a programmer instructs a computer to perform certain operations. The other extremal view is *mathematical-logical* in which the programming language is a medium of expressing abstract computational concepts such as algorithms or data structures. Historically, the first programming languages were, by necessity, machine-oriented, but algorithmic (i.e. mathematical-logical) machine independent languages appeared soon after (FORTRAN, LISP, ALGOL, etc.). The case for machine independent programming has been made, quite successfully, a long time ago.

Machine independence means that programs can just be recompiled to run on different devices with similar architectures. *Architecture independence* pushes this idea even further to devices with *different* architectures, such as conventional CPUs, distributed systems, GPUs and reconfigurable hardware. In a series of papers [12–14] we have examined the possibility of lifting machine independence to architecture independence in the context of distributed computing. The reason is that in distributed computing the *deployment* of a program is often reflected at the level of source code. For example, these two ERLANG programs:

```
c(A_pid)-> receive X -> A_pid ! X*X end, c(A_pid).
main()->
  C_pid = spawn(f, c, [self()]), C_pid ! 3,
  receive X -> C_pid ! 4, receive Y -> X+Y end
end.
```

and

```
c()-> receive {Pid, X} -> Pid ! X*X end, c().
b(A_pid, C_pid)-> receive
    request0 -> C_pid ! {self(), 3},
                receive X -> A_pid ! X end;
    request1 -> C_pid ! {self(), 4},
                receive X -> A_pid ! X end end,
  b(A_pid, C_pid).
main()->
  C_pid = spawn(f, c, []),
  B_pid = spawn(f, b, [self(), C_pid]),
  B_pid ! request0, receive X ->
  B_pid ! request1, receive Y -> X+Y end end.
```

perform the same function, which in a computation on a single node we may write as `let f = λx. x * x in f 3 + f 4`, except that the deployment is different. The first program is distributed on two nodes whereas the second is distributed on three nodes. Our proposal is as follows:

> *The program should not need to specify details of the runtime deployment pattern.*

This means that communication and process management should largely disappear from the source code, as they are the means by which deployment patterns are implemented. To explain our metaphor, they are the *seam* which we aim to eliminate. For example, the deployment pattern could be indicated using a configuration file or pragma-like code annotations assigning nodes to arbitrary sub-terms of the program:

```
let f = (λx. x * x)@C in (f 3 + f 4)@B
```


*ICFP'14*, September 1–6, 2014, Gothenburg, Sweden.

ACM 978-1-4503-2873-9/14/09…$15.00.
http://dx.doi.org/10.1145/2628136.2628152

We would like the invocations of `f` in the program to semantically work exactly like a local function call even though the function is located on the node `C` and invoked from node `B`. The compiler should automatically handle the communication for us by for triggering a small exchange of messages between the two nodes: For example, node `B` would start by sending a message to `C`, requesting the evaluation of `f` and providing the location of the argument and where to return to.

At this early stage we make no claims that our approach is a practical alternative to established distributed programming methodologies, but we believe that there should be room for exploring machine (or architecture) independent, purely algorithmic, languages in the context of distributed computing. Our approach is in contrast to communication-oriented idioms such as MPI and ERLANG. It is also philosophically different to *domain specific languages* for distributed computing, which expose communication but use concepts associated with high-level programming languages such as *types* [17] to avoid certain classes of errors. The closest to our approach are *remote procedure calls*, so our research programme can be reformulated as an answer to the question:

> *Can remote procedure calls be incorporated transparently, correctly and efficiently into the programming language?*

### 1.1 Contribution

In this paper we present an extension of the classic Krivine abstract machine for the call-by-name lambda calculus [19] for seamless distributed computing. Our previous work gave a compilation technique [13] based on the Geometry of Interaction (GOI) token abstract machine [23] and another one [14] based on game semantics (GAMC) [15]. They both achieve seamless distribution but have certain apparent unavoidable inefficiencies. The GOI-based compilation has the potential to be locally efficient but has a possibly insurmountable communication overhead, whereas the games-based compiler communicates efficiently but requires very high computational overhead on each node. The current approach, which we are also exploring for the SECD machine and call-by-value [12], combines the best of both worlds: it communicates efficiently by keeping the size of the message within a small fixed bound and it executes efficiently on each node. In fact, the compilation scheme degenerates to that of the conventional Krivine machine if the whole program is deployed on a single node. An additional advantage which this current technique offers is that, unlike the exotic GOI and games-based approaches, it is in some technical sense standard so that it can interface trivially with legacy code which was compiled to the Krivine machine.

Because the formal definitions of the formalism of *Krivine Nets* which we propose can be in places fairly intricate we adopt a fully formal approach, expressing all the definitions and the correctness proofs in the dependently-typed programming language AGDA [27]. This allows us to present technical results with a high degree of confidence and to remove all proof details, which can be found elsewhere [1], from the paper. In this paper we can focus on the exposition. The reader is not assumed to know AGDA in order to read the paper, which is self-contained, but a good knowledge of the language is required in order to understand the correctness proofs.

## 2. The Krivine machine

We are compiling the untyped applied call-by-name lambda calculus, i.e. a lambda calculus with constants. For the sake of a concrete yet simple presentation we assume that the only data is natural numbers, and the constants are numeric literals, arithmetic operators and if-then-else. Informally, the grammar of the language is

$$M ::= x \mid \lambda x.M \mid MM \mid \mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M \mid n \mid M \oplus M \mid M@A.$$

Formally, we define the data-type of *terms* with the following constructors:

```
data Term : ⋆ where
  λ_    : Term → Term
  _$_   : (t t' : Term) → Term
  var   : ℕ → Term
  lit   : ℕ → Term
  op    : (f : ℕ → ℕ → ℕ) (t t' : Term) → Term
  if0_then_else_ : (b t f : Term) → Term
  _@_ : Term → Node → Term
```

Above, $\star$ is the "type of types". We are using the De Bruijn index notation, so abstraction (**λ_**) is a unary operator and each variable (**var**) is a natural number. The value of the index denotes the number of binders between the variable and its binder. Function application (**_$_**, an *infix* operator) is an explicit constructor, for clarity. Numeric literals (**lit**) and branching (**if0_then_else_**) are obvious, noting that the constructor for the latter is a *mixfix operator*. Binary arithmetic operators (**op**) take three arguments: the function giving the operation and two terms.

We also introduce syntactic support in the language (**_@_**, another infix operator) for specifying node assignments for *closed* sub-terms. This is done strictly for simplicity. Node assignment could be otherwise specified, e.g. using a separate configuration file, but it would needlessly complicate the presentation. Node assignment is a "compiler pragma" and has no bearing on observational properties of the programming languages. The requirement that node assignment is specified for closed terms only keeps the presentation as simple as possible. This apparent restriction can be easily overcome using lambda lifting.

EXAMPLE 1. *The term* $(\lambda x.\lambda y.y + x)\,3\,4$ *is represented as*

```
termExample : Term
termExample = λ (λ (var 0 +' var 1)) $ lit 3 $ lit 4
  where _+'_ = op _+_
```

The Krivine machine is the standard abstract machine for call-by-name. It has three components: code, environment and stack. The stack and the environment contain *thunks*, which are closures representing unevaluated function arguments. The evaluations are delayed until the values are needed. For the pure lambda calculus, the Krivine machine uses three instructions:

**POPARG** pop an argument from the stack and add it to the environment.

**PUSHARG** push a thunk for some code given as argument.

**VAR** look up the argument in the environment and start evaluation.

For the applied lambda calculus the machine becomes more complex because arithmetic operations are strict, so extra mechanisms are required to force the evaluation of arguments.

Formally, we define closures and environments by mutual recursion:

```
mutual
  Closure = Term × Env
  data EnvEl : ⋆ where
    clos : Closure → EnvEl
  Env = List EnvEl
```

The constructor **clos** that takes a *Closure* into an environment element *EnvEl* is only needed for formal reasons, to prevent the AGDA type-checker from reporting a circular definition.

Stacks and configurations are:

```
data StackElem : ⋆ where
  arg  : Closure                        → StackElem
  if0  : Closure          → Closure → StackElem
  op₂  : (ℕ → ℕ → ℕ) → Closure → StackElem
  op₁  : (ℕ → ℕ)                        → StackElem
Stack  = List StackElem
Config = Term × Env × Stack
```

The generic stack elements (for function arguments) are constructed using **arg**, whereas **if0**, $\mathbf{op}_2$, $\mathbf{op}_1$ are used by the constants.

The signature of the Krivine machine is given as a data-type, defining a *Rel*ation on *Config*urations of the Krivine machine:

```
data _ ⟶_ _ : Rel Config Config where
```

The relation type *Rel A B* is defined to be $A \to B \to \star$, so two elements $a$ and $b$ are $R$-related exactly when $R\ a\ b$ is inhabited, given $R : Rel\ A\ B$. Each rule, i.e. each instruction of the machine, will thus correspond to a constructor. We explain the formal definition of each rule.

$$\mathbf{POPARG} : \{t : Term\}\ \{e : Env\}\ \{c : Closure\}\ \{s : Stack\} \to (\boldsymbol{\lambda}\ t, e, \mathbf{arg}\ c :: s) \longrightarrow_{\mathcal{K}} (t, \mathbf{clos}\ c :: e, s)$$

**POPARG** handles abstractions $\boldsymbol{\lambda}\ t$ by moving the top of the stack **arg** $c$ into the first position of the environment $e$. The constructors **arg**, **clos** are needed for type-checking and would be omitted in an informal presentation. The constructor arguments ($t$, $e$, $c$, $s$) are implicit, indicated syntactically in AGDA by curly brackets.

$$\mathbf{PUSHARG} : \{t\ t' : Term\}\ \{e : Env\}\ \{s : Stack\} \to ((t\ \$\ t'), e, s) \longrightarrow_{\mathcal{K}} (t, e, \mathbf{arg}\ (t', e) :: s)$$

**PUSHARG** handles application $t\ \$\ t'$ by creating a new closure **arg** $(t', e)$ and pushing it onto the stack, then carrying on with the execution of the function body $t$.

$$\mathbf{VAR} : \{n : \mathbb{N}\}\ \{e\ e' : Env\}\ \{t : Term\}\ \{s : Stack\} \to lookup\ n\ e \equiv \mathbf{just}\ (\mathbf{clos}\ (t, e')) \to (\mathbf{var}\ n, e, s) \longrightarrow_{\mathcal{K}} (t, e', s)$$

In AGDA the $\equiv$ operator denotes *propositional* equality, which necessitates a proof, whereas $=$ is used to introduce new definitions. The **VAR** rule looks up the variable $n$ in the current environment $e$ and, if successful, retrieves the closure at that position $(t, e')$ and proceeds to execute from it, with the current stack.

Because this is an applied lambda calculus we need additional operations for conditionals and operators. Here we omit the types of the implicit arguments since they can be inferred:

$$\begin{aligned}
&\mathbf{COND} : \forall\ \{b\ t\ f\ e\ s\} \to (\mathbf{if0}\ b\ \mathbf{then}\ t\ \mathbf{else}\ f, e, s) \longrightarrow_{\mathcal{K}} (b, e, \mathbf{if0}\ (t, e)\ (f, e) :: s)\\
&\mathbf{COND\text{-}0} : \forall\ \{e\ t\ e'\ f\ s\} \to (\mathbf{lit}\ 0, e, \mathbf{if0}\ (t, e')\ f :: s) \longrightarrow_{\mathcal{K}} (t, e', s)\\
&\mathbf{COND\text{-}suc} : \forall\ \{n\ e\ t\ f\ e'\ s\} \to (\mathbf{lit}\ (1 + n), e, \mathbf{if0}\ t\ (f, e') :: s) \longrightarrow_{\mathcal{K}} (f, e', s)\\
&\mathbf{OP} : \forall\ \{f\ t\ t'\ e\ s\} \to (\mathbf{op}\ f\ t\ t', e, s) \longrightarrow_{\mathcal{K}} (t, e, \mathbf{op}_2\ f\ (t', e) :: s)\\
&\mathbf{OP}_2 : \forall\ \{n\ e\ f\ t\ e'\ s\} \to (\mathbf{lit}\ n, e, \mathbf{op}_2\ f\ (t, e') :: s) \longrightarrow_{\mathcal{K}} (t, e', \mathbf{op}_1\ (f\ n) :: s)\\
&\mathbf{OP}_1 : \forall\ \{n\ e\ f\ s\} \to (\mathbf{lit}\ n, e, \mathbf{op}_1\ f :: s) \longrightarrow_{\mathcal{K}} (\mathbf{lit}\ (f\ n), [\,], s)
\end{aligned}$$

EXAMPLE 2. *We can see the Krivine machine at work in this simple example. The term in Ex. 1 has the following execution trace, written informally as follows:*

$$\begin{aligned}
&((\boldsymbol{\lambda}\ (\boldsymbol{\lambda}\ _{+}_\ 0\ 1)\ \$\ 3\ \$\ 4), [\,], [\,])\\
&\quad\longrightarrow\langle\ \mathbf{PUSHARG}\ \rangle\\
&((\boldsymbol{\lambda}\ (\boldsymbol{\lambda}\ _{+}_\ 0\ 1)\ \$\ 3), [\,], [(4, [\,])])\\
&\quad\longrightarrow\langle\ \mathbf{PUSHARG}\ \rangle\\
&(\boldsymbol{\lambda}\ (\boldsymbol{\lambda}\ _{+}_\ 0\ 1), [\,], [(3, [\,]), (4, [\,])])\\
&\quad\longrightarrow\langle\ \mathbf{POPARG}\ \rangle\\
&(\boldsymbol{\lambda}\ _{+}_\ 0\ 1, [(3, [\,])], [(4, [\,])])\\
&\quad\longrightarrow\langle\ \mathbf{POPARG}\ \rangle\\
&(_{+}_\ 0\ 1, [(4, [\,]), (3, [\,])], [\,])\\
&\quad\longrightarrow\langle\ \mathbf{OP}\ \rangle\\
&(0, [(4, [\,]), (3, [\,])], [\mathbf{op}_2\ _{+}_\ (1, [(4, [\,]), (3, [\,])])])\\
&\quad\longrightarrow\langle\ \mathbf{VAR\ refl}\ \rangle\\
&(4, [\,], [\mathbf{op}_2\ _{+}_\ (1, [(4, [\,]), (3, [\,])])])\\
&\quad\longrightarrow\langle\ \mathbf{OP}_2\ \rangle\\
&(1, [(4, [\,]), (3, [\,])], [\mathbf{op}_1\ (_{+}_\ 4)])\\
&\quad\longrightarrow\langle\ \mathbf{VAR\ refl}\ \rangle\\
&(3, [\,], [\mathbf{op}_1\ (_{+}_\ 4)])\\
&\quad\longrightarrow\langle\ \mathbf{OP}_1\ \rangle\\
&(7, [\,], [\,])
\end{aligned}$$

*In the above we have omitted the constructors* ***op, var, arg****, etc. for brevity.*

Finally, we include a (degenerate) instruction for remote execution:

$$\mathbf{REMOTE} : \forall\ \{t\ i\ e\ s\} \to (t\ @\ i, e, s) \longrightarrow_{\mathcal{K}} (t, [\,], s)$$

This instruction is included strictly so that the $\_@\_$ construct for node assignment does not trigger a runtime error, but it is effectively a *no-op*: it simply erases the environment $e$, since node assignment is meant to be applied only to closed terms. In the following section we will define the distributed Krivine machine, where the **REMOTE** instruction is meaningful.

## 3. Krivine nets

### 3.1 The machine

We now extend the Krivine machine so that it supports an arbitrary pattern of distribution by letting several instances of the extended machine run in a network. We call these machines *DKrivine* machines and they form *Krivine Nets*. The DKrivine machines extend the Krivine machines conservatively by adding new features. Each such machine is identified as a *node* in the network and has a dedicated heap. A pointer into a heap may be tagged with a node identifier, case in which it is a *remote pointer*, which can now be stored in the environment along with local closures. The stack may now have as a bottom element a remote pointer indicating the existence of a *remote stack extension*, i.e. the fact that the information which logically belongs to this stack is physically located on a different node. Finally, the configuration of the Krivine machine is now called a *thread* indicating that its execution can be dynamically started and halted. Internally, the heap structure is used for storing persistent data that needs to out-live the runtime of a thread. The new formal definitions are as follows:

```
RPtr   = Ptr × Node
ContPtr = RPtr

data EnvElem : ⋆ where
  local  : Closure →          EnvElem
  remote : ContPtr → ℕ → EnvElem

Stack    = List StackElem × Maybe (ContPtr × ℕ × ℕ)
ContHeap = Heap Stack
Thread   = Term × Env × Stack
Machine  = Maybe Thread × ContHeap
```

The definitions are straightforward, except for the **remote** environment element and the definition of stacks which require explanation. A remote *ContPtr* is a pointer to a continuation stack, and the constructor **remote** takes an additional natural number argument indicating the offset in that continuation stack where the referred closure is stored. As stated, the stack now possibly includes a remote

stack extension. This extension is to be thought of as being located at the bottom of the local stack, and consists of a *ContPtr* pointing into the heap of a remote node holding the stack, and two natural numbers that form the current node's *view* of that stack. The second number is the offset into the remote stack that the view starts from, and the first number stores how many consecutive arguments there are on on it.

Because DKrivine machines are networked they exchange messages, which fall into three categories, formalised as constructors for the *Msg* datatype:

**REMOTE** A message with this tag initiates remote evaluation, formally defined as

$$\textbf{REMOTE} \;:\; \mathit{Term} \to \mathit{Node} \to \mathit{ContPtr} \to \mathbb{N} \to \mathit{Msg}$$

The message consists of a *Term*, a destination *Node* identifier, a *ContPtr* to the sender's current continuation stack and a natural number indicating how many arguments are on that stack.

The design decision to make a *Term* part of the message structure is for simplicity of formalisation only. In the actual implementation only a *code pointer* needs to be sent to the node, which already has the required code available. The mechanism through which compiled code arrives at each node is handled by a *distributed program loader* (see e.g. [11]) which is part of the runtime system and, as such, beyond the scope of this work. It should be obvious that distributed program loading is possible in principle here because all code is static and available at compile-time.

**RETURN** These messages are sent when computation has terminated and reached a literal, and the value must be returned to the node that has initiated the computation. The formal definition is:

$$\textbf{RETURN} \;:\; \mathit{ContPtr} \to \mathbb{N} \to \mathbb{N} \to \mathit{Msg}$$

The message contains a *ContPtr* to the remote stack of the machine that is receiving the message, the natural number calculated and another number indicating to the receiving machine how many arguments can be now discarded from the stack, corresponding to the offset of sending node's view of the stack.

**VAR** is a message used to access remotely located variables. It consists of a remote *ContPtr*, an offset into the remote continuation stack, a local continuation stack and the number of arguments on it.

$$\textbf{VAR} \;:\; \mathit{ContPtr} \to \mathbb{N} \to \mathit{ContPtr} \to \mathbb{N} \to \mathit{Msg}$$

We need to send the continuation stack of the calling node (like in the **REMOTE** rule) because the remote variable may refer to a function, in which case the arguments are supplied by the calling node, or it may be part of an operation on the calling node, in which case the resulting number needs to be returned there once it has been calculated.

Deliberate in the design of the Krivine nets is the need to minimise message exchange. To achieve this, machines do not send remote "pop" messages for manipulating remote stack extensions, but perform this operation locally. When a node sends a pointer to a new continuation stack it also sends the number of arguments that are on that stack, so that the receiving node can pop arguments from its local view of that stack.

We can now start describing the transitions of the DKrivine machine. The signature of the transition relation is:

$$\mathsf{data}\ \_ \vdash \longrightarrow_{\mathcal{DK}} \langle\_\rangle\_\ (i \;:\; \mathit{Node}) \;:\; \mathit{Machine} \to \mathit{Tagged\ Msg} \to \mathit{Machine} \to \star$$

DKrivine transitions are parameterised by the current node identifier and map a *Machine* state and a *Tagged Msg* into a new *Machine* state. The tag applied to the message indicates whether the message is sent, received or absent (i.e. a *silent* transition):

$$\begin{array}{l}\mathsf{data}\ \mathit{Tagged}\ (\mathit{Msg} \;:\; \star) \;:\; \star\ \mathsf{where}\\ \quad \boldsymbol{\tau} \quad\quad : \mathit{Tagged\ Msg}\\ \quad \textbf{send} \quad : \mathit{Msg} \to \mathit{Tagged\ Msg}\\ \quad \textbf{receive} \;:\; \mathit{Msg} \to \mathit{Tagged\ Msg}\end{array}$$

All the old rules are present, but now expressed in the presence of the continuation heap.

$$\begin{array}{l}\textbf{POPARG} \;:\; \forall\,\{t\ e\ c\ s\ r\ ch\} \to\\ \quad i \vdash (\textbf{just}\ (\boldsymbol{\lambda}\ t, e, \textbf{arg}\ c :: s, r), ch) \;\longrightarrow_{\mathcal{DK}} \langle\, \boldsymbol{\tau} \,\rangle\\ \quad\quad (\textbf{just}\ (t, \textbf{local}\ c :: e, s, r), ch)\end{array}$$

Compared to the **POPARG** rule of the original machine, the only differences are the tag on the configuration (**just** ...), which expresses the fact that the DKrivine thread is running, and the continuation heap *ch* which remains constant during the application of this rule. The environment element constructor **local** now emphasises that the variable is local. Because the transition involves only one node it is $\boldsymbol{\tau}$, i.e. no messages are exchanged.

The other old transition rules are embedded into the DKrivine machine in a similar way. They are all silent and the continuation heap *ch* stays unchanged:

$$\begin{array}{l}\textbf{PUSHARG} \;:\; \forall\,\{t\ t'\ e\ s\ r\ ch\} \to\\ \quad i \vdash (\textbf{just}\ ((t\ \$\ t'), e, s, r), ch) \;\longrightarrow_{\mathcal{DK}} \langle\, \boldsymbol{\tau} \,\rangle\\ \quad\quad (\textbf{just}\ (t, e, \textbf{arg}\ (t', e) :: s, r), ch)\\ \textbf{VAR} \;:\; \forall\,\{n\ e\ s\ r\ ch\ t\ e'\} \to\\ \quad \mathit{lookup}\ n\ e \equiv \textbf{just}\ (\textbf{local}\ (t, e')) \to\\ \quad i \vdash (\textbf{just}\ (\textbf{var}\ n, e, s, r), ch) \;\longrightarrow_{\mathcal{DK}} \langle\, \boldsymbol{\tau} \,\rangle\\ \quad\quad (\textbf{just}\ (t, e', s, r), ch)\\ \textbf{COND} \;:\; \forall\,\{b\ t\ f\ e\ s\ r\ ch\} \to\\ \quad i \vdash (\textbf{just}\ (\textbf{if0}\ b\ \textbf{then}\ t\ \textbf{else}\ f, e, s, r), ch) \;\longrightarrow_{\mathcal{DK}} \langle\, \boldsymbol{\tau} \,\rangle\\ \quad\quad (\textbf{just}\ (b, e, \textbf{if0}\ (t, e)\ (f, e) :: s, r), ch)\\ \textbf{COND-0} \;:\; \forall\,\{e\ t\ e'\ f\ s\ r\ ch\} \to\\ \quad i \vdash (\textbf{just}\ (\textbf{lit}\ 0, e, \textbf{if0}\ (t, e')\ f :: s, r), ch) \;\longrightarrow_{\mathcal{DK}} \langle\, \boldsymbol{\tau} \,\rangle\\ \quad\quad (\textbf{just}\ (t, e', s, r), ch)\\ \textbf{COND-suc} \;:\; \forall\,\{n\ e\ t\ e'\ f\ s\ r\ ch\} \to\\ \quad i \vdash (\textbf{just}\ (\textbf{lit}\ (1 + n), e, \textbf{if0}\ t\ (f, e') :: s, r), ch) \;\longrightarrow_{\mathcal{DK}} \langle\, \boldsymbol{\tau} \,\rangle\\ \quad\quad (\textbf{just}\ (f, e', s, r), ch)\\ \textbf{OP} \;:\; \forall\,\{f\ t\ t'\ e\ s\ r\ ch\} \to\\ \quad i \vdash (\textbf{just}\ (\textbf{op}\ f\ t\ t', e, s, r), ch) \;\longrightarrow_{\mathcal{DK}} \langle\, \boldsymbol{\tau} \,\rangle\\ \quad\quad (\textbf{just}\ (t, e, \textbf{op}_2\ f\ (t', e) :: s, r), ch)\\ \textbf{OP}_2 \;:\; \forall\,\{n\ e\ f\ t\ e'\ s\ r\ ch\} \to\\ \quad i \vdash (\textbf{just}\ (\textbf{lit}\ n, e, \textbf{op}_2\ f\ (t, e') :: s, r), ch) \;\longrightarrow_{\mathcal{DK}} \langle\, \boldsymbol{\tau} \,\rangle\\ \quad\quad (\textbf{just}\ (t, e', \textbf{op}_1\ (f\ n) :: s, r), ch)\\ \textbf{OP}_1 \;:\; \forall\,\{n\ e\ f\ s\ r\ ch\} \to\\ \quad i \vdash (\textbf{just}\ (\textbf{lit}\ n, e, \textbf{op}_1\ f :: s, r), ch) \;\longrightarrow_{\mathcal{DK}} \langle\, \boldsymbol{\tau} \,\rangle\\ \quad\quad (\textbf{just}\ (\textbf{lit}\ (f\ n), [\,], s, r), ch)\end{array}$$

The **REMOTE** execution rule is now meaningful, and it has a **send** and a **receive** version:

$$\begin{array}{l}\textbf{REMOTE-send} \;:\; \forall\,\{t\ i'\ e\ s\ ch\} \to\\ \quad \mathsf{let}\ (ch', kp) \;=\; i \vdash ch \rhd s\ \mathsf{in}\\ \quad i \vdash (\textbf{just}\ (t\ @\ i', e, s), ch)\\ \quad\quad \longrightarrow_{\mathcal{DK}} \langle\, \textbf{send}\ (\textbf{REMOTE}\ t\ i'\ kp\ (\mathit{num\text{-}args}\ s)) \,\rangle\\ \quad\quad (\textbf{nothing}, ch')\end{array}$$

The operation $i \vdash ch \rhd s$ signifies allocating at node $i$ in heap $ch$ a new pointer pointing at stack $s$, and it returns a pair of the updated heap $ch'$ and the newly allocated remote pointer $kp$. The remote-execution directive $t\ @\ i'$ is carried out by sending a **REMOTE** message to $i'$ consisting of the (pointer to) code $t$, the destination $i'$, the local continuation-stack pointer $kp$ and the number of arguments on it. After sending the remote execution message the thread halts, i.e. its state is **nothing**.

The function that calculates the number of arguments on the stack is quite subtle and we give its formal expression below:

$$
\begin{array}{lll}
\mathit{num\text{-}args} \ : \ \mathit{Stack} & & \to \mathbb{N} \\
\mathit{num\text{-}args} \ (\,[\,] & , \mathbf{nothing}) & = \ 0 \\
\mathit{num\text{-}args} \ (\,[\,] & , \mathbf{just} \ (\_, n, \_)) & = \ n \\
\mathit{num\text{-}args} \ (\mathbf{arg} \ \_ :: s & , r) & = \ 1 + \mathit{num\text{-}args} \ (s, r) \\
\mathit{num\text{-}args} \ (\mathbf{if0} \ \_ \ \_ :: \_ & , \_) & = \ 0 \\
\mathit{num\text{-}args} \ (\mathbf{op}_2 \ \_ \ \_ :: \_, \_) & & = \ 0 \\
\mathit{num\text{-}args} \ (\mathbf{op}_1 \ \_ :: \_ & , \_) & = \ 0
\end{array}
$$

The function returns the number of arguments at the top of the stack, but it takes into account the possibility that some arguments are local and some arguments are remote. Recall that the remote pointer that we store at the bottom of the stack, pointing to the remote stack extension, also has a natural number *numargs* expressing how many arguments are stored remotely. This is an important optimisation because it makes it possible for this function to be evaluated *locally*, without querying the remote machine where the stack extension is physically located.

The counterpart **REMOTE-receive** rule is:

$$
\begin{array}{l}
\mathbf{REMOTE\text{-}receive} \ : \ \forall \ \{ch \ t \ kp \ numargs\} \to \\
\quad i \vdash (\mathbf{nothing}, ch) \\
\qquad \longrightarrow_{\mathcal{DK}} \langle \, \mathbf{receive} \ (\mathbf{REMOTE} \ t \ i \ kp \ numargs) \, \rangle \\
\qquad (\mathbf{just} \ (t, [\,], [\,], \mathbf{just} \ (kp, numargs, 0)), ch)
\end{array}
$$

The thread on node $i$ is halted when it receives the **REMOTE** execution message, with the same contents as above. The code $t$ becomes the currently executed code, in an empty environment ($t$ is, as we explained before, closed) and empty stack remotely extended by $kp$ to the originating machine stack.

Additionally, some of the original rules now have *send* and *receive* counterparts to handle the situation when remote variables or continuations need to be processed. Remarkably, it is possible to avoid sending messages when popping a remote argument, and we can get by with the following new instruction:

$$
\begin{array}{l}
\mathbf{POPARG\text{-}remote} \ : \ \forall \ \{t \ e \ kp \ args \ m \ ch\} \to \\
\quad i \vdash (\mathbf{just} \ (\boldsymbol{\lambda} \ t, e, [\,], \mathbf{just} \ (kp, 1 + \ args, m)), ch) \\
\qquad \longrightarrow_{\mathcal{DK}} \langle \, \boldsymbol{\tau} \, \rangle \\
\qquad (\mathbf{just} \ (t, \mathbf{remote} \ kp \ m :: e, [\,], \mathbf{just} \ (kp, args, 1 + \ m)), ch)
\end{array}
$$

Note that this is a silent ($\boldsymbol{\tau}$) transition. A machine does not really "pop" the arguments of a remote stack extension but changes its view of this remote stack. This avoids instituting a whole class of messages for stack management and it also gives a more robust stack management framework in which stacks, along with heaps and any other data structures involved, are only changed *locally*.

This rule is triggered when a **POPARG** action encounters a local empty stack, which means that the remote stack extension needs to be used. Just like in the case of a local **POPARG**, the environment is updated, but this time with the remote pointer $kp$ which has its offset set at $m$. The offset in the view of the remote stack extension is updated (to $1 + \ m$) to reflect the fact that another argument has been "popped".

The rules that need genuine remote counterparts are **VAR**, for accessing remote variables, and **RETURN**, for returning a literal from a remote computation.

$$
\begin{array}{l}
\mathbf{VAR\text{-}send} \ : \ \forall \ \{n \ e \ s \ rkp \ index \ ch\} \to \\
\quad \mathit{lookup} \ n \ e \equiv \mathbf{just} \ (\mathbf{remote} \ rkp \ index) \to \\
\quad \mathsf{let} \ (ch', kp) \ = \ i \vdash ch \rhd s \ \mathsf{in} \\
\quad i \vdash (\mathbf{just} \ (\mathbf{var} \ n, e, s), ch) \\
\qquad \longrightarrow_{\mathcal{DK}} \langle \, \mathbf{send} \ (\mathbf{VAR} \ rkp \ index \ kp \ (\mathit{num\text{-}args} \ s)) \, \rangle \\
\qquad (\mathbf{nothing}, ch')
\end{array}
$$

The rule is triggered when the machine detects a **remote** pointer in its environment $e$. Just like in the case of the **REMOTE** instruction, the current continuation stack is saved in the continuation heap of the machine $i$, at address $kp$. The machine then sends a **VAR**-tagged message onto the network, with the structure discussed before, and halts, i.e. its thread is **nothing**. Note that the left-hand-side of the transition triggered by the **VAR-send** rule is almost the same as that of the local **VAR** rule.

Upon receiving a **VAR** message, a (halted) machine executes the **VAR-receive** instruction:

$$
\begin{array}{l}
\mathbf{VAR\text{-}receive} \ : \ \forall \ \{ch \ kp \ s \ n \ rkp \ m \ el\} \to \\
\quad ch \ ! \ kp \equiv \mathbf{just} \ s \to \\
\quad \mathit{stack\text{-}index} \ s \ n \equiv \mathbf{just} \ el \to \\
\quad i \vdash (\mathbf{nothing}, ch) \\
\qquad \longrightarrow_{\mathcal{DK}} \langle \, \mathbf{receive} \ (\mathbf{VAR} \ (kp, i) \ n \ rkp \ m) \, \rangle \\
\qquad (\mathbf{just} \ (\mathbf{var} \ 0, el :: [\,], [\,], \mathbf{just} \ (rkp, m, 0)), ch)
\end{array}
$$

The right-hand-side of the **VAR-receive** rule introduces a new variable **var** *0*, perhaps surprisingly. In order to avoid having special cases where the retrieved variable index is itself either local or remote, we create the dummy variable **var** *0* referring to the variable pointed-to by the received **VAR** message. This is what the $\mathit{stack\text{-}index} \ : \ \mathit{Stack} \to \mathbb{N} \to \mathit{Maybe \ EnvElem}$ function, invoked on the stack that $kp$ points to, achieves. If the stack element at index $n$ in the stack is a local argument, then it returns that closure as a **local** environment element. If the element at index $n$ refers to an argument on the remote stack extension, it returns a corresponding **remote** environment element. Afterwards we can use the existing local **VAR** or **VAR-send** rules depending on whether the variable is local or remote also to this node.

$$
\begin{array}{l}
\mathbf{RETURN\text{-}send} \ : \ \forall \ \{n \ e \ kp \ m \ ch\} \to \\
\quad i \vdash (\mathbf{just} \ (\mathbf{lit} \ n, e, [\,], \mathbf{just} \ (kp, 0, m)), ch) \\
\qquad \longrightarrow_{\mathcal{DK}} \langle \, \mathbf{send} \ (\mathbf{RETURN} \ kp \ n \ m) \, \rangle \\
\qquad (\mathbf{nothing}, ch) \\
\mathbf{RETURN\text{-}receive} \ : \ \forall \ \{ch \ kp \ s \ s' \ n \ m\} \to \\
\quad ch \ ! \ kp \equiv \mathbf{just} \ s \to \mathit{drop\text{-}stack} \ s \ m \equiv \mathbf{just} \ s' \to \\
\quad i \vdash (\mathbf{nothing}, ch) \\
\qquad \longrightarrow_{\mathcal{DK}} \langle \, \mathbf{receive} \ (\mathbf{RETURN} \ (kp, i) \ n \ m) \, \rangle \\
\qquad (\mathbf{just} \ (\mathbf{lit} \ n, [\,], s'), ch)
\end{array}
$$

Finally, the **RETURN-send** and **RETURN-receive** rules are triggered when a machine has reached a literal and has a remote stack extension without any arguments, implying that the remote stack is either empty (i.e. it is located at the root node of the whole execution) or it has a continuation requiring a natural number literal. In both cases we want to send the literal back to the node where the stack is located. The one thing to notice is that the message includes the number $m$ to be used by the receiver to drop the correct number of elements from the top of the stack. This is handled by the *drop-stack* function, defined as follows:

$$
\begin{array}{lll}
\mathit{drop\text{-}stack} \ : \ \mathit{Stack} \to \mathbb{N} \to \mathit{Maybe \ Stack} & & \\
\mathit{drop\text{-}stack} \ (s, r) & 0 & = \ \mathbf{just} \ (s, r) \\
\mathit{drop\text{-}stack} \ ([\,], \mathbf{just} \ (\_, 0, \_)) & (1 + \ \_) & = \ \mathbf{nothing} \\
\mathit{drop\text{-}stack} \ ([\,], \mathbf{just} \ (kp, 1 + \ n, m)) & (1 + \ i) & = \\
\quad \mathit{drop\text{-}stack} \ ([\,], \mathbf{just} \ (kp, n, 1 + \ m)) \ i & & \\
\mathit{drop\text{-}stack} \ ([\,], \mathbf{nothing}) & (1 + \ \_) & = \ \mathbf{nothing} \\
\mathit{drop\text{-}stack} \ (\mathbf{arg} \ \_ :: s, r) & (1 + \ i) & = \ \mathit{drop\text{-}stack} \ (s, r) \ i \\
\mathit{drop\text{-}stack} \ (\_ :: \_, \_) & (1 + \ \_) & = \ \mathbf{nothing}
\end{array}
$$

As in the case of *num-args* the function may change the local view of a remote stack extension, without requiring further message exchanges between nodes. If not enough arguments are on the stack the function returns **nothing**, which should not happen during a normal execution since we take care to keep the stack views consistent.

### 3.2 The network

We consider two kinds of networks, either based on synchronous message passing (blocking send) or asynchronous message passing (non-blocking send). The two definitions are:

$$
\begin{array}{lll}
\mathcal{SN}et & = & \mathit{Node} \to \mathit{Machine} \\
\mathcal{AN}et & = & (\mathit{Node} \to \mathit{Machine}) \times \mathit{List \ Msg}
\end{array}
$$

The way we model the asynchronous network is inspired by the Chemical Abstract Machine (CHAM) [3]. The network is, in addition to a family of machines indexed by *Node* identifiers, a global multiset of messages *List Msg* in which sent messages

```
data _ ⟶_S_ (nodes : SNet) : SNet → ⋆ where
  silent-step : ∀ {i m'} → (i ⊢ nodes i ⟶ ⟨τ⟩ m') → nodes ⟶_S update nodes i m'
  comm-step : ∀ {s r msg sender' receiver'} →
    let nodes' = update nodes s sender' in
    (s ⊢ nodes s ⟶ ⟨send msg⟩ sender') → (r ⊢ nodes' r ⟶ ⟨receive msg⟩ receiver') → nodes ⟶_S update nodes' r receiver'
data _ ⟶_As_ : AsNet → AsNet → ⋆ where
    step : ∀ {nodes} msgsl msgsr {tmsg m' i} →
      let (msgin, msgout) = detag tmsg in
      (i ⊢ nodes i ⟶ ⟨tmsg⟩ m') → (nodes, msgsl ++ msgin ++ msgsr) ⟶_As (update nodes i m', msgsl ++ msgout ++ msgsr)
```

**Figure 1.** Network transitions

are placed, and from which received messages are retrieved. The formal definitions are given in Fig. 1.

In the $\mathcal{S}Net$ messages are passed directly between machines. Network transitions are either a **silent-step** when a node makes a $\tau$ transition, or **comm-step** when two nodes exchange information. The $\mathcal{A}s\mathcal{N}et$ only has a generic **step**, because no synchronisation is needed. A machine on a node may take a $\tau$ step or a communication step, case in which a message is placed or removed from the global set of messages. The function *detag* figures out what messages a node is sending and receiving, allowing one rule for all three cases, as at most one of *msgin* and *msgout* in the rule is non-empty:

```
detag : {A : ⋆} → Tagged A → List A × List A
detag τ          = [] , []
detag (send x)   = [] , [x]
detag (receive x) = [x], []
```

Another helper function used in the definitions is *update*, which updates the state of a node in the network. It is the usual function update, commonly written as $(f \mid x \mapsto y)$, here relying on the assumption that the set of node identifiers has decidable equality ($\_\stackrel{?}{=}\_$). It is formally defined as:

```
update : {A : ⋆} → (Node → A) → Node → A → Node → A
update nodes n m n' with n' ≟ n
update nodes n m n'      | yes _ = m
update nodes n m n'      | no _  = nodes n'
```

In AGDA, the with keyword introduces patterns additional to the arguments in a function definition.

The definition of network transitions is parameterised by a machine transition relation $\_ \vdash \_ \longrightarrow \langle\_\rangle \_$, which is subsequently instantiated to $\longrightarrow_{\mathcal{DK}}$, and initialised by starting from a designated node *i* with code *t* and all other constituents empty.

```
open import Network Node _≟_ _ ⊢⟶_DK ⟨_⟩_ public
initial-network_S : Term → Node → SNet
initial-network_S t i =
  update (λ i' → (nothing, ∅)) i (just (t, [], [], nothing), ∅)
initial-network_As : Term → Node → AsNet
initial-network_As c i = initial-network_S c i, []
```

It is immediate to show that a $\mathcal{S}Net$ can be represented by the more expressive $\mathcal{A}s\mathcal{N}et$. This is the function mapping a *Sync* transition to an *Async* one by placing, then removing, the message in the global message pool (here $\_^+$ takes the transitive closure of a relation, constructed with list-like notation):

```
Sync-to-Async+ : ∀ {a b} → (a ⟶_S b) →
                           (a, []) ⟶+_As (b, [])
Sync-to-Async+ (silent-step s)      = [step [] [] s]
Sync-to-Async+ (comm-step s1 s2) = step [] [] s1 :: [step [] [] s2]
```

The other direction is not as trivial, and is formalised by the following lemma, stating that whenever some DKrivine machines can make an *Async* transition with the global pool of messages remaining the same (empty, for simplicity), the same transition could be made in a $\mathcal{S}Net$:

```
Async+-to-Sync+ : ∀ {nodes nodes'} i →
                  all nodes except i are inactive →
                  ((nodes, []) ⟶+_As (nodes', [])) →
                  nodes ⟶+_S nodes'
Async+-to-Sync+ = Async+-to-Sync+-lemma refl refl
```

The proof is an immediate application of a more complex lemma which is omitted from this presentation. In contrast to the *Sync-to-Async*$^+$ embedding, this embedding is specific to DKrivine machines. More precisely, two properties of these machines make this possible: The first one is that the DKrivine machines halt after each message **send** and **receive** only from halting states. The second one is that they are deterministic. Intuitively, it is fairly clear that the two styles of communication are equivalent under these circumstances.

These two results about Krivine Nets are interesting because they show that we do not need to commit to a synchronous or asynchronous network of DKrivine machines since they are equivalent. We may therefore use whichever is more convenient for correctness proofs in the knowledge that the properties we prove transfer immediately to the other one.

### 3.3 Example

Let us compare briefly the execution of a rather simple term,

$$((\lambda f.\lambda x.f\ x)@B)(\lambda y.y)\ 0$$

on a single machine and on a distributed machine. The program is located on (the default) node *A*, except for $\lambda f.\lambda x.f\ x$ which is on node *B*. This program is similar to our introductory example in that it does a remote function call, and additionally shows that higher-order remote function calls are also possible.

As we discussed earlier, the Krivine machine ignores the @ construct (the **REMOTE** rule is a no-op), producing the execution trace **PUSHARG**; **PUSHARG**; **REMOTE**; **POPARG**; **POPARG**; **PUSHARG**; **VAR**; **POPARG**; **VAR**; **VAR**, which leaves the machine in state (**lit** *0*, [], []).

The Krivine Net of two nodes produces the following trace (informally, indicating machine state only when interesting). Node *A* starts with **PUSHARG**; **PUSHARG**; **REMOTE-send**, which produces message

**REMOTE** (**λ** (**λ** (**var** *1* **$** **var** *0*))) *B* ($ptr_1$, *A*) *2*

where $ptr_1$ points to the stack ([(**λ** **var** *0*, []), (*0*, [])], **nothing**).

Node *B* receives the message and executes **REMOTE-receive**; **POPARG-remote**; **POPARG-remote**; **PUSHARG**; **VAR-send**, which produces message **VAR** ($ptr_1$, *A*) *0* ($ptr_2$, *B*) *1*, where $ptr_2$ points to the stack

[(**var** *0*, [**remote** ($ptr_1$, *A*) *1*, **remote** ($ptr_1$, *A*) *0*])], **just** (($ptr_1$, *A*), *0*, *2*)

Note that the two traces are essentially the same, except for the **REMOTE** rule becoming meaningful. As we explained before, the **POPARG-remote** rule only changes the local view of the remote stack extension and generates no communication overhead. Also note that the stack at $ptr_2$ extends remotely to the stack at $ptr_1$ and uses it in its own stored closures.

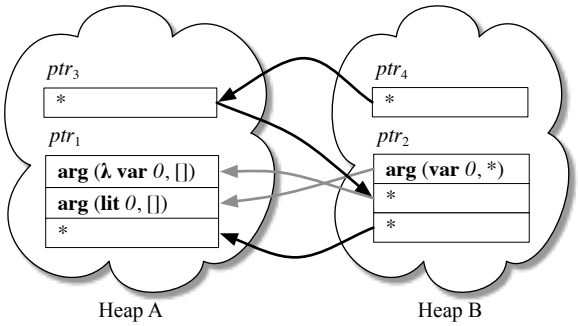


**Figure 2.** Final heap

The rest of the dialogue is as follows:

*Node A* : **VAR-receive**; **VAR**; **POPARG-remote**; **VAR-send**
*Node B* : **VAR-receive**; **VAR**; **VAR-send**
*Node A* : **VAR-receive**; **VAR**; **RETURN-send**
*Node B* : **RETURN-receive**; **RETURN-send**
*Node A* : **RETURN-receive**; **RETURN-send**
*Node B* : **RETURN-receive**; **RETURN-send**
*Node A* : **RETURN-receive**

Compared to the Krivine trace, the **VAR** instructions is here broken into a **send** and **receive** version if the requested variable is remote. There is also the additional **VAR** rule needed to avoid a case statement on whether a variable is local or remote. The **RETURN** instructions are new, required to forward computed values to the caller.

After the execution, the heaps of the two nodes are:

$$A : \{ptr_1 \mapsto ([\textbf{arg}\ (\boldsymbol{\lambda}\ \textbf{var}\ 0, [\,]); \textbf{arg}\ (\textbf{lit}\ 0, [\,])], \textbf{nothing}),\ ptr_3 \mapsto ([\,], \textbf{just}\ ((ptr_2, B), 0, 1))\}$$
$$B : \{ptr_2 \mapsto ([\textbf{arg}\ (\textbf{var}\ 0, [\textbf{remote}\ (ptr_1, A)\ 1; \textbf{remote}\ (ptr_1, A)\ 0])], \textbf{just}\ ((ptr_1, A), 0, 2)),\ ptr_4 \mapsto ([\,], \textbf{just}\ ((ptr_3, A), 0, 0))\}$$

A graphical representation of the final heap is in Fig. 2, with stack extension pointers in black and remote variables in grey.

Unlike the Krivine machine, the Krivine Nets will result in non-empty heaps (*garbage*) in the individual DKrivine machines. We will discuss how to deal with this in the conclusion.

## 4. Correctness

We prove the correctness of the DKrivine Net by exhibiting a simulation between the conventional Krivine machine and a Krivine Net. The simulation is then used to prove the following *Soundness theorems*:

$$\textit{termination-agrees}_{\mathcal{S}} : \forall\ \textit{cfg nodes}\ n \to R_{\mathcal{S}}\ \textit{cfg nodes} \to \textit{cfg} \downarrow_{\mathcal{K}} \textbf{lit}\ n \to \textit{nodes} \downarrow_{\mathcal{S}} \textbf{lit}\ n$$
$$\textit{divergence-agrees}_{\mathcal{S}} : \forall\ \textit{cfg nodes} \to R_{\mathcal{S}}\ \textit{cfg nodes} \to \textit{cfg} \uparrow_{\mathcal{K}} \to \textit{nodes} \uparrow_{\mathcal{S}}$$

The termination theorem states that for any Krivine machine configuration *cfg* and any Krivine Net configuration *nodes*, if we have a *simulation relation* $R_{\mathcal{S}}$ between them then for any literal *n*, if the Krivine machine starting from *cfg* produces the literal *n*, then the Krivine Net starting from configuration *nodes* produces the same. Note that we are using *Sync* nets, because they are more convenient and because *Async* nets can be reduced to *Sync* nets in the case of Krivine Nets, as discussed in Sec. 3.2. The divergence theorem makes a similar point about non-termination: from related states, if the Krivine machine diverges then the Krivine Net diverges.

### 4.1 The simulation relation

The most important ingredient of the correctness proof is defining and exhibiting the appropriate simulation relation. At the top level, the relation between the Krivine machine and Krivine Nets configurations is defined formally as follows:

$$R_{\mathcal{S}} : \textit{Rel Config}\ \mathcal{S}Net$$
$$R_{\mathcal{S}}\ \textit{cfg nodes} = \exists\ \lambda\ i \to \textit{all nodes except i are inactive} \times R_{\textit{Machine}}\ (\textit{proj}_2 \circ \textit{nodes})\ \textit{cfg}\ (\textit{proj}_1\ (\textit{nodes}\ i))$$

In AGDA notation the existential statement $\exists i.P(i)$ is written $\exists\ \lambda\ i \to P\ i$. The predicate *all_except_are_* is defined as

$$\textit{all f except x are P} = \forall\ x' \to x' \not\equiv x \to P\ (f\ x')$$

and *inactive node* holds when the thread of *node* is **nothing**. A simulation between machine and net configurations exists only when precisely one node *i* is active in the net. The machine at that node ($\textit{proj}_1$ (*nodes i*)) must be related to the configuration of the Krivine machine through the following machine-simulation relation:

$$R_{\textit{Machine}} : \textit{Heaps} \to \textit{Rel Config}\ (\textit{Maybe Thread})$$
$$R_{\textit{Machine}}\ \textit{hs}\ (t_1, e_1, s_1)\ (\textbf{just}\ (t_2, e_2, s_2)) = R_{\textit{Term}}\ t_1\ t_2 \times R_{\textit{Env}}\ \textit{hs}\ e_1\ e_2 \times (\exists\ \lambda\ \textit{rank} \to R_{\textit{Stack}}\ \textit{rank hs}\ s_1\ s_2)$$
$$R_{\textit{Machine}}\ \textit{hs}\ (t_1, e_1, s_1)\ \textbf{nothing} = \bot$$

The relation is indexed by the distributed heap of the Krivine Net *hs* : *Heaps*, which is the *Node*-indexed family of all the individual heaps. This relation $R_{\textit{Machine}}$ simply distributes the relation further to terms using $R_{\textit{Term}}$, environments using $R_{\textit{Env}}$ and stacks using $R_{\textit{Stack}}$. In order for this to be possible it is required that the DKrivine machine is not halted (**nothing** : *Maybe Thread*).

On terms, the relation $R_{\textit{Term}}$ is just propositional equality, while $R_{\textit{Env}}$ and $R_{\textit{Stack}}$ are more subtle and require a non-trivial proof technique. $R_{\textit{Stack}}$ is similar to a *step-indexed* relation [2] on stacks. It is defined by induction on a natural number *rank* in order to ensure that the cascading remote stack extensions do not have any cycles. Unlike a step-indexed relation, *rank* means that we do exactly *rank* remote-pointer dereferencings in the process of relating two stacks, and $R_{\textit{Stack}}$ requires that this number is known. $R_{\textit{EnvElem}}$, used by $R_{\textit{Env}}$ to relate environment elements, is defined by induction on a *rank* for the same reason.

#### 4.1.1 Relating environments

On environments, the formal definition of the relation is:

$$R_{\textit{Env}} : \textit{Heaps} \to \textit{Rel Krivine.Env DKrivine.Env}$$
$$R_{\textit{Env}}\ \textit{hs}\ [\,]\ [\,] = \top$$
$$R_{\textit{Env}}\ \textit{hs}\ [\,]\ (x_2 :: e_2) = \bot$$
$$R_{\textit{Env}}\ \textit{hs}\ (x_1 :: e_1)\ [\,] = \bot$$
$$R_{\textit{Env}}\ \textit{hs}\ (x_1 :: e_1)\ (x_2 :: e_2) = (\exists\ \lambda\ \textit{rank} \to R_{\textit{EnvElem}}\ \textit{rank hs}\ x_1\ x_2) \times R_{\textit{Env}}\ \textit{hs}\ e_1\ e_2$$

Empty environments are trivially related, but environments of different shapes cannot be related. If both environments are non-empty then the definition is inductive on the structure of the environment. Environment elements are related by requiring that there exists a *rank* such that they are related by $R_{\textit{EnvElem}}$:

$$R_{\textit{EnvElem}} : \mathbb{N} \to \textit{Heaps} \to \textit{Rel Krivine.EnvElem DKrivine.EnvElem}$$
$$R_{\textit{EnvElem}}\ 0\ \textit{hs}\ (\textbf{clos}\ c_1)\ (\textbf{local}\ c_2) = R_{\textit{Closure}}\ \textit{hs}\ c_1\ c_2$$
$$R_{\textit{EnvElem}}\ (1 + \textit{rank})\ \textit{hs}\ (\textbf{clos}\ c_1)\ (\textbf{local}\ c_2) = \bot$$
$$R_{\textit{EnvElem}}\ 0\ \textit{hs}\ (\textbf{clos}\ c_1)\ (\textbf{remote}\ \textit{contptr index}) = \bot$$
$$R_{\textit{EnvElem}}\ (1 + \textit{rank})\ \textit{hs}\ (\textbf{clos}\ c_1)\ (\textbf{remote}\ \textit{contptr index}) = \textit{stack-ext-pred hs contptr}\ (\lambda\ s_2 \to \exists\ \lambda\ ee_2 \to \textit{stack-index}\ s_2\ \textit{index} \equiv \textbf{just}\ ee_2 \times R_{\textit{EnvElem}}\ \textit{rank hs}\ (\textbf{clos}\ c_1)\ ee_2)$$

Local closures of the DKrivine machine relate to closures of the Krivine machine ($R_{\textit{Closure}}$) if their terms are equal and their environments are related through $R_{\textit{Env}}$. Relating remote closures

(**remote** *contptr index*) of the DKrivine machine to the closures of the Krivine machine (**clos** $c_1$) is perhaps the most subtle part of the definition. It uses the following helper function which ensures that, given a distributed heap $hs : Heaps$, a remote pointer $(ptr, loc) : ContPtr$ and a predicate on distributed stacks $DKrivine.Stack \to \star$, the pointer points to a stack in the heap of node $loc$ such that the predicate holds:

$$\begin{array}{l} stack\text{-}ext\text{-}pred : Heaps \to ContPtr \to (DKrivine.Stack \to \star) \to \star \\ stack\text{-}ext\text{-}pred\ hs\ (ptr, loc)\ P = \exists\ \lambda\ s \to hs\ loc\ !\ ptr \equiv \mathbf{just}\ s \times P\ s \end{array}$$

The pointer dereferencing operation is $hs\ loc\ !\ ptr$. The predicate which we use in the definition of $R_{EnvElem}$ is that there exists an element $ee_2$ in the environment of the DKrivine machine such that it $R_{EnvElem}$ relates to the Krivine closure **clos** $c_1$ in one less step.

Note that the *rank* has to be *0* to relate local elements, and it has to be $1 +$ *rank* to relate a remote element. The recursive call is done with the predecessor *rank*, which makes sure that there are exactly *rank* pointers to follow to reach a local closure if we have an element of $R_{EnvElem}\ rank\ hs\ x_1\ x_2$.

### 4.1.2 Relating stacks

Relating stacks is somewhat similar.

$$\begin{array}{lll} R_{Stack} : \mathbb{N} \to Heaps \to Rel\ Krivine.Stack\ DKrivine.Stack & & \\ R_{Stack}\ rank & hs\ (x_1 :: s_1)\ ([], \mathbf{nothing}) & = \bot \\ R_{Stack}\ rank & hs\ [\,]\ (x_2 :: s_2, r) & = \bot \\ R_{Stack}\ 0 & hs\ [\,]\ ([], \mathbf{nothing}) & = \top \\ R_{Stack}\ (1 + rank)\ hs\ [\,] & ([], \mathbf{nothing}) & = \bot \\ R_{Stack}\ rank & hs\ (x_1 :: s_1)\ (x_2 :: s_2, r) & = \\ \quad R_{StackElem}\ hs\ x_1\ x_2 \times R_{Stack}\ rank\ hs\ s_1\ (s_2, r) & & \\ R_{Stack}\ 0 & hs\ s_1\ ([], \mathbf{just}\ (contptr, args, drop)) & = \bot \\ R_{Stack}\ (1 + rank)\ hs\ s_1\ ([], \mathbf{just}\ (contptr, args, drop)) & & = \\ \quad stack\text{-}ext\text{-}pred\ hs\ contptr\ (\lambda\ s_2 \to & & \\ \quad\quad \exists\ \lambda\ ds_2 \to drop\text{-}stack\ s_2\ drop \equiv \mathbf{just}\ ds_2 & & \\ \quad\quad\quad \times\ num\text{-}args\ ds_2 \equiv args \times R_{Stack}\ rank\ hs\ s_1\ ds_2) & & \end{array}$$

Empty stacks, with no remote extensions, are related if the *rank* is 0, whereas empty and non-empty are not. Two non-empty stacks are related if the elements on top are related by $R_{EnvElem}$ and the remaining stacks are related. The relation is interesting when remote pointer extensions are involved. If there is a remote stack extension but the step index is 0 then it cannot be related to a Krivine stack. If there is a non-zero step index then, using the same helper function *stack-ext-pred*, we require that the sub-stack $ds_2$ of $s_2$ obtained by dropping the *drop* arguments required by the remote stack extension pointer **just** $(contptr, args, drop)$ is related to the Krivine stack $s_1$ using a smaller (by one) index.

Finally, stack elements are related if they have the same head constructor, and the constituents are related:

$$\begin{array}{lll} R_{StackElem} : Heaps \to Rel\ Krivine.StackElem\ DKrivine.StackElem & & \\ R_{StackElem}\ hs\ (\mathbf{arg}\ c_1) & (\mathbf{arg}\ c_2) & = R_{Closure}\ hs\ c_1\ c_2 \\ R_{StackElem}\ hs\ (\mathbf{if0}\ c_1\ c_1')\ (\mathbf{if0}\ c_2\ c_2') & & = R_{Closure}\ hs\ c_1\ c_2 \times \\ & & \quad R_{Closure}\ hs\ c_1'\ c_2' \\ R_{StackElem}\ hs\ (\mathbf{op}_2\ f\ c_1) & (\mathbf{op}_2\ g\ c_2) & = f \equiv g \times \\ & & \quad R_{Closure}\ hs\ c_1\ c_2 \\ R_{StackElem}\ hs\ (\mathbf{op}_1\ f) & (\mathbf{op}_1\ g) & = f \equiv g \\ R_{StackElem}\ hs\ \_ & \_ & = \bot \end{array}$$

## 4.2 Proof outline

In order to prove the main property we need to first establish the monotonicity of all the heap-indexed relations relative to heap inclusion: if two configurations, machines, environments, environment elements or stacks are related in a family of heaps *hs* they are also related in any larger family of heaps $hs\ \subseteq_s\ hs'$. The properties are proved in a module parameterised by the heap inclusion property, and therefore it does not need to be included in each statement – it is a background assumption:

$$\begin{array}{l} \mathsf{module}\ HeapUpdate\ (hs\ hs' : Heaps)\ (inc : hs \subseteq_s hs')\ \mathsf{where} \\ \quad envelem : \forall\ rank\ el\ el' \to R_{EnvElem}\ rank\ hs\ el\ el' \\ \qquad\qquad \to R_{EnvElem}\ rank\ hs'\ el\ el' \\ \quad env : \forall\ e\ e' \to R_{Env}\ hs\ e\ e' \to R_{Env}\ hs'\ e\ e' \\ \quad stackelem : \forall\ el\ el' \to R_{StackElem}\ hs\ el\ el' \\ \qquad\qquad \to R_{StackElem}\ hs'\ el\ el' \\ \quad stack : \forall\ rank\ s\ s' \to R_{Stack}\ rank\ hs\ s\ s' \\ \qquad\qquad \to R_{Stack}\ rank\ hs'\ s\ s' \\ \quad machine : \forall\ cfg\ m \to R_{Machine}\ hs\ cfg\ m \\ \qquad\qquad \to R_{Machine}\ hs'\ cfg\ m \end{array}$$

The proofs are largely straightforward, inductively on the structure of the data structure the lemma is concerned with. The key auxiliary property that makes monotonicity of the relations true is the fact that any predicate which relies on heap dereferencing is preserved:

$$\begin{array}{l} s\text{-}ext\text{-}pred : \forall\ contptr\ \{P\ Q\} \to (\forall\ s \to P\ s \to Q\ s) \to \\ \quad stack\text{-}ext\text{-}pred\ hs\ contptr\ P \to stack\text{-}ext\text{-}pred\ hs'\ contptr\ Q \end{array}$$

For example, for environments, environment elements and closures the proofs are mutually recursive, inductive on their structures:

$$\begin{array}{l} closure : \forall\ c\ c' \to R_{Closure}\ hs\ c\ c' \to R_{Closure}\ hs'\ c\ c' \\ envelem : \forall\ rank\ el\ el' \to R_{EnvElem}\ rank\ hs\ el\ el' \\ \quad \to R_{EnvElem}\ rank\ hs'\ el\ el' \\ envelem\ 0 \quad (\mathbf{clos}\ c)\ (\mathbf{local}\ c')\ Rcc' = closure\ c\ c'\ Rcc' \\ envelem\ (1 + rank)\ (\mathbf{clos}\ c)\ (\mathbf{local}\ c')\ Rcc' = Rcc' \\ envelem\ 0 \quad (\mathbf{clos}\ c)\ (\mathbf{remote}\ contptr\ index)\ Relel' = Relel' \\ envelem\ (1 + rank)\ (\mathbf{clos}\ c)\ (\mathbf{remote}\ contptr\ index)\ Relel' = \\ \quad s\text{-}ext\text{-}pred\ contptr\ f\ Relel' \\ \quad \mathsf{where} \\ \quad\quad f : \forall\ s \to \\ \quad\quad (\exists\ \lambda\ ee' \to stack\text{-}index\ s\ index \equiv \mathbf{just}\ ee' \\ \quad\quad\quad \times R_{EnvElem}\ rank\ hs\ (\mathbf{clos}\ c)\ ee') \to \\ \quad\quad\quad \exists\ \lambda\ ee' \to stack\text{-}index\ s\ index \equiv \mathbf{just}\ ee' \\ \quad\quad\quad \times R_{EnvElem}\ rank\ hs'\ (\mathbf{clos}\ c)\ ee' \\ \quad\quad f\ s\ (ee', si, Rcee') = ee', si, envelem\ rank\ (\mathbf{clos}\ c)\ ee'\ Rcee' \\ env : \forall\ e\ e' \to R_{Env}\ hs\ e\ e' \to R_{Env}\ hs'\ e\ e' \\ env\ [\,]\ [\,]\ Ree' = Ree' \\ env\ [\,]\ (x :: e')\ Ree' = Ree' \\ env\ (x :: e)\ [\,]\ Ree' = Ree' \\ env\ (x :: e)\ (x' :: e')\ ((rank, Rxx'), Ree') = \\ \quad (rank, envelem\ rank\ x\ x'\ Rxx'), env\ e\ e'\ Ree' \\ closure\ (t, e)\ (t', e')\ (Rtt', Ree') = Rtt', env\ e\ e'\ Ree' \end{array}$$

The soundness theorem $termination\text{-}agrees_{\mathcal{S}}$ stated at the beginning of this section follows directly from two important lemmas, $simulation_{\mathcal{S}}$ and *termination-return*. The former is the main technical result of the paper (soundness is merely a corollary of it) and the latter is used to handle the only non-trivial case of the soundness proof, that of cascading **RETURN** statements at the end of an execution.

The theorem

$$simulation_{\mathcal{S}} : Simulation\ \_ \longrightarrow_{\mathcal{K}} \_\ \_ \longrightarrow^{+}_{\mathcal{S}} \_\ R_{\mathcal{S}}$$

states that $R_{\mathcal{S}}$, discussed in the previous sub-section, is a *Simulation* relation between the $\longrightarrow_{\mathcal{K}}$ and $\longrightarrow^{+}_{\mathcal{S}}$ transition relations. The simulation relation is defined in the standard way, where $\longrightarrow$ and $\longrightarrow'$ are transition relations:

$$\begin{array}{l} Simulation : (\_R\_ : Rel\ A\ B) \to \star \\ Simulation\ \_R\_ = \forall\ a\ a'\ b \to (a \longrightarrow a') \to a\ R\ b \to \\ \quad\quad \exists\ \lambda\ b' \to (b \longrightarrow' b') \times a'\ R\ b' \end{array}$$

The proof of $simulation_{\mathcal{S}}$ is lengthy but largely routine. The non-trivial cases are:

- **RETURN** actions of the DKrivine machines, which are handled by the lemma *simulation-return*:

```
simulation-return : ∀ n e s cfg' e' s' i nodes srank conth →
  let  cfg = (lit n, e, s)
       hs = proj₂ ∘ nodes
    in cfg ⟶_K cfg' →
       all nodes except i are inactive →
       nodes i ≡ just (lit n, e', s'), conth →
       R_Stack srank hs s s' → ∃ λ nodes' →
         nodes ⟶⁺_S nodes' × R_S cfg' nodes'
```

- **VAR** remote actions of the DKrivine machine, which are handled by the lemma *simulation-var*:

```
simulation-var : ∀ t e s n e' s' nodes i conth el →
  let hs = proj₂ ∘ nodes in
    (∃ λ rank → R_EnvElem rank hs (clos (t, e)) el) →
    (∃ λ rank → R_Stack rank hs s s') →
    all nodes except i are inactive →
    nodes i ≡ just (var n, e', s'), conth →
    lookup n e' ≡ just el →
    ∃ λ nodes' → (nodes ⟶⁺_S nodes') × R_S (t, e, s) nodes'
```

What is interesting about these two lemmas, which establish the conditions under which the simulation relation is preserved by transitions related to the integer operations and **VAR** rules, is that it requires a different proof technique, induction on the *rank*. This is because the distributed machine may need to perform a cascade of returns (or variable accesses) between different nodes before it reaches a configuration related to that of the Krivine machine, as we saw in the example in Sec. 3.3.

The *termination-return* lemma mentioned earlier uses a similar proof technique (induction on the *rank*); its full statement is:

```
termination-return : ∀ n e' s' i nodes srank conth →
  let hs = proj₂ ∘ nodes
    in all nodes except i are inactive →
       nodes i ≡ just (lit n, e', s'), conth →
       R_Stack srank hs [] s' → nodes ↓_S lit n
```

The second part of the soundness proof is the agreement on divergence between the Krivine machine and the Krivine net. This proof relies essentially on the fact that a Krivine Net transition is deterministic whenever only one node is active and that the Krivine machine transition's codomain is decidable in the following sense (proofs are omitted):

```
_is-deterministic-at_ : {A B : ⋆} (R : Rel A B) (x : A) → ⋆
_R_ is-deterministic-at a = ∀ {b b'} → a R b → a R b' → b ≡ b'

determinism_S : ∀ nodes i → all nodes except i are inactive →
  _ ⟶_S _ is-deterministic-at nodes

_is-decidable : {A B : ⋆} (_R_ : Rel A B) → ⋆
_R_ is-decidable = ∀ a → Dec (∃ λ b → a R b)

decidable_K : _ ⟶_K _ is-decidable
```

To conclude this section, we also need to show that initial configurations are related so that we have a starting point for the simulation. This is easy to prove since the environments and stacks are empty:

```
initial-related_S : ∀ t root → R_S (t, [], []) (initial-network_S t root)
```

## 5. Proof of concept implementation

We have implemented a prototype compiler for Krivine Nets [1]. Except for the _@_ directive, compilation to Krivine Nets is implemented by using the same standard compilation scheme used to compile to Krivine machines. It is the runtime system of the DKrivine machine that takes into account whether pointers are local or remote and behaves in the correct way. The _@_ directive is translated directly into a predefined **REMOTE** bytecode instruction, which constructs and sends a **REMOTE** message at runtime. As we said, we avoid sending code by grouping fragments of output code that correspond to the same node, and compiling each group as a separate binary. The fragment of code that corresponds to $t$ inside a subterm $t$ @ $A$ is assigned, at compile-time, a global identifier that an invoking node can use to activate $t$ on node $A$, meaning that no actual code has to be sent at runtime.

The "bytecode" of the Krivine machine is translated into C functions, and message passing is implemented using MPI. The aim is not efficiency as much as simplicity. The compiler is not certified or extracted from the proofs, so we choose an implementation that is, as much as reasonably possible, "clearly correct."

### 5.1 Comparison with GOI and GAMC

A principled comparison with the GOI and GAMC compilers, which also follow the methodological principles of seamless compilation, is difficult because it cannot be a precise like-for-like comparison. We summarise the differences between the three implementations below.

Krivine Nets and GOI implement call-by-name PCF, but not in the same way. Krivine Nets implement the type-free language and recursion is dealt with using a Y combinator in the source programming language, which is inefficient. On the other hand, the GOI compiler uses a specialised fixpoint constant but it also requires specialised machinery to handle variable contraction. So there are various contingent sources of inefficiencies which straightforward but laborious optimisations could remove.

The GAMC compiler, on the other hand, implements a much larger language: a typed applied CBN lambda calculus with mutable references and concurrency. It is also tidier than the Krivine Net approach, in that it explicitly deallocates useless memory. These features require a significant amount of overhead, some of which already is present in the DKrivine infrastructure but some of which will need to be subsequently added.

However, there are some features that make the comparison of the three compilers meaningful. The first one is that the programs we use are virtually identical and the fact that they all use CBN means no language-level considerations come into play. The second one is that all three compilers are written as representations of the semantic model of the language, with a similar level of disregard for optimisations against a similar level of concern for "obvious" correctness. The third one is that all three target C and MPI, meaning that benchmarks can be run on the same computer.

With these significant caveats in mind we will attempt a rough performance comparison of the three compilers in several ways. Our benchmarks are small programs operating on integers:

`arith`**:** Computing the sum of applying a complicated integer function to the numbers in the sequence $0, \ldots, 299$.

`fib`**:** Computing the 10th Fibonacci number (using the exponential algorithm) 100 times and taking the sum.

`root`**:** Compute the (integer) root of a polynomial using 20 iterations of the bisection method.

***Krivine baseline.*** We take the classic Krivine machine as a reference point and run the compilers in a degenerate single-node mode. This gives a rough measure of the overall overhead of the compiler before communication costs even come into play. In the case of the GOI compiler the overheads are mainly due to the implementation of contraction, whereas in the case of GAMC they are due to the large amount of heap allocation and deallocation. However, note that the discrepancy would be even greater if we had a fixpoint operator in the Krivine machine instead of relying on the term-level Y combinator.

| | arith | fib | root |
|---|---|---|---|
| Baseline | 100% (0.34s) | 100% (0.094s) | 100% (0.009s) |
| GOI | 3,042% (10.3s) | 2,832% (2.7s) | 20,222% (0.18s) |
| GAMC | 765% (2.6s) | 395% (0.53s) | 356% (0.032s) |
| DKrivine | 131% (0.44s) | 141% (0.13s) | 233% (0.021s) |

***Single node baseline.*** We measure each compiler using its own single-node performance as a reference point and we split the program in two nodes such that a large communication overhead is introduced. We measure it both in terms of relative execution time and in terms of average and maximum size of the messages, in bytes. Note that the overheads are only due to the processing required by the node to send and receive the nodes and not due to network latencies. In order to factor them out we run all the (virtual) MPI nodes on the same physical computer.

The data is shown in Tab. 1 and we can see that the DKrivine compiler is not only faster for local execution, but also has a comparatively small communication overhead. Each time entry in the table is relative to the same compiler's local execution time, and the absolute time is shown in parentheses. We can see that DKrivine is well ahead of the others in terms of absolute execution time. Both GAMC and DKrivine use messages of a bound size, whereas GOI's messages grow, sometimes significantly, during execution. The high overhead across all three compilers for the `root` benchmark is because it does a relatively small amount of local computations before it needs to communicate. We suspect that the high overhead for GOI and GAMC in many benchmarks is also due to the large amount of "bookkeeping" C code that is required, even for simple terms. The way the C compiler optimiser works plays an important role in the performance gap between single node and distribution. When all the code is on the same node the functions are aggressively inlined because they belong to the same binary output. When the code is distributed this is no longer possible. Also, an analysis of the produced code shows that the C optimiser generally struggles with the code for the distributed nodes, because it does not have a view of the whole program.

## 6. Previous and related work

Programming languages and libraries for distributed and client-server computing (which can be seen as a particularly simple form of distribution) are a vast area of research. Relevant to us are functional programming languages for distributed execution, and several surveys are available [21, 32].

Functional programming languages for distributed systems take different approaches in terms of process and communication management. Languages such as ERLANG, which are meant for system-level development offer a fairly low-level view of distribution in which both process and communication are managed explicitly; this is the language we used for contrasting effect in the introduction. To tame communication some languages in this category use mechanisms imported from process-calculi, such as PICT [33]. Programming languages do not need to be created from scratch to include improved language support for communication. Session types have been used to extend a variety of languages, including functional languages, with better communication primitives [34] or, alternatively, to provide language-independent frameworks for integrating distributed applications, such as SCRIBBLE [1].

Our approach is, however, quite different. We aim to make communication implicit, or *seamless*. In some sense this is already widely used in programming practice, especially in the context of client-server applications, in the form of *remote procedure calls* (RPC) and related technologies such as *Simple Object Access Protocol* (SOAP). What we aim to do is to integrate these approaches into the programming language so that from a programmer perspective there is no distinction between a remote and local call, even at higher order. Perhaps the closest to our aim is Remote Evaluation (REV) [31], another generalisation of RPC, which enables the use of higher-order functions across node boundaries. The main differences between REV and our work is that REV relies on sending unevaluated code. The REV approach evolved into a variety of *mobile code languages* [7] which add several layers of sophistication to this approach, but have evolved in a direction that is not directly relevant to transparent distribution.

The EDEN project [22], an implementation of parallel HASKELL for distributed systems which keeps most communication implicit, is also close to our aims. Another similarity to our work is that the specification of the language is tiered: an operational semantics at the level of the language and an abstract-machine semantics for execution environment, *the Distributed Eden Abstract Machine* (DREAM) [5]. EDEN is not perfectly seamless: a small set of syntactic constructs are used to manage processes explicitly and communication is always performed using head-strict lazy lists. There are significant technical differences between DREAM and Krivine Nets since the DREAM is a mechanism of distribution for the *Spineless Tagless G-machine* [18] whereas we develop the Krivine machine. Also, in terms of emphasis, EDEN is an implementation-focussed project whereas we want to create a firm theoretical foundation on which compilation to distributed platforms can be carried out. Whereas (as far as we know) no soundness results exist for the DREAM, we provide a fully formalised proof.

Other similar implementation-oriented projects are for *tierless client-server computing* such as LINKS [8], where "tierless" has a similar meaning to our use of "seamless". The execution mechanism that LINKS builds on, the client/server calculus [9], is specialised to systems with two nodes, namely client and server. The two nodes are not equal peers: the server is designed to be *stateless* to be able to handle a large number of clients. The work on the client/server calculus also spawned work on a more general parallel abstract machine, LSAM, that handles an arbitrary number of nodes [26]. A predecessor to LSAM, called DML, uses a similar abstract machine but for a richer language [28]. The main difference between these machines and Krivine Nets is that they are based on higher-level machines for call-by-value lambda calculi, that use explicit substitutions and are therefore less straightforward to use as a basis for compilation. In contrast to our work, they also assume synchronous communication models.

Abstract machines for distributed systems have also been studied. In fact, as early as 1980 a formal proposal for standardising distributed computing using an abstract machine model was put forth, although it did not catch on [30]. The DREAM, DML and the LSAM are, as far as we are aware, the only abstract machines for general distributed systems which, like the DKrivine machine, combine conventional execution mechanisms with communication primitives. Abstract machines for communication only have been proposed [16], inspired by the CHAM (which we also take inspiration from, to model the communication network), but they only deal with half the problem when it comes to compiling conventional languages.

Finally, we mention the compilation of conventional programming languages to (possibly) distributed architectures via process calculi, such as PICT [33], which also uses an abstract machine with communication primitives. We have studied techniques based on interaction semantics in prior work, using the Geometry of Interaction [13] or Game Semantics [14]. Although such more exotic approaches can be effective at creating correct and transparent distribution, it seems to be the case that the single-node execution model is bound to be less efficient than that of conventional abstract machines. Without over-emphasising efficiency at this early stage,

[1] `https://www.jboss.org/scribble`

| | arith | | | fib | | | root | | |
|---|---|---|---|---|---|---|---|---|---|
| | time | avg. size | max. size | time | avg. size | max. size | time | avg. size | max size |
| GOI | 114% (11.6s) | 107 | 172 | 4,017% (3.8s) | 302 | 444 | 19,422% (1.7s) | 717 | 1,312 |
| GAMC | 193% (5.0s) | 20 | 24 | 1,481% (7.8s) | 20 | 24 | 22,872% (7.3s) | 20 | 24 |
| DKrivine | 140% (0.62s) | 32 | 40 | 238% (0.32s) | 32 | 40 | 890% (0.19s) | 32 | 40 |

**Table 1.** Benchmarks for distribution overheads

it is also the case that interfacing code compiled using conventional techniques with code compiled using exotic techniques is difficult and leads to problems with interoperability via foreign-function interfaces. To us this is a significant short-coming which the current work seeks to avoid.

## 7. Conclusion

In this paper we have presented a method of distributing the execution of the Krivine machine into what we call a Krivine Net. This gives us a principled compilation model of the applied CBN lambda calculus to an abstract distributed architecture. Our main results are a rigorous, fully formalised, proof of correctness of the Krivine Net by comparing it to the conventional Krivine machine, and a proof-of-concept compiler which allows us to compare this compilation scheme with alternative methods based on other abstract machines and more exotic semantics such as Geometry of Interaction and Games. Compared to the more implementation-oriented prior work on transparent (or tierless or seamless) compilation to distributed or client-server architectures, our emphasis is on correctness. We believe that our main contribution is a theoretical firm starting point for the principled study of compilation targetting such architectures.

A broader question worth asking is whether this transparent and integrated approach to distributed computing is *practical*. There are two main possible objections:

**Performance** Some might say that higher-level languages have poorer performance than system-oriented programming language, which makes them impractical. This debate started when Backus proposed FORTRAN as a machine-independent programming language, and has carried on fruitlessly ever since. We believe that the full spectrum of languages, from machine code to the most abstract, are worth investigating seriously. Seamless computing focusses on the latter, somewhat in the extreme, in the belief that the principled study of heterogeneous (not just distributed, but also reconfigurable etc.) compilation techniques will broaden and deepen our understanding of programming languages in general. And, if we are lucky and diligent, it may even yield a practical and very useful programming paradigm.

**Control** Distributed computing raises certain specific obstacles in the way of using higher-level languages seamlessly, and this leads to more cogent arguments against their use. A distributed architecture is more volatile than a single node because individual nodes may fail, communication links may break and messages may get lost. Because of this, a remote call may fail in ways that a local call may not. Is it reasonable to present them to the programmer as if they are the same thing? We argue that there is a significant class of applications where the answer is *yes*. If the programmer's objectives are algorithmic rather than the development of systems, it does not seem right to burden them with the often onerous task of failure management in a distributed system. Another argument against higher-level languages is that they may hide the details of the program's dataflow and not provide enough control to eliminate bottlenecks. To us it seems that the right way to manage both failure and dataflow issues in distributed *algorithmic* programming requires a separation of concerns. Suitable runtime systems must present a more robust programming interface; MAPREDUCE [10] and CIEL [25] are examples of execution engines with runtime systems that automatically handle configuration and failure management aspects, the latter supporting dynamic dataflow dependencies. If more fine-grained control is required, then separate deployment and configuration policies which are transparent to the programmer should be employed. In general, we believe that the role and the scope of orchestration languages [6] should be greatly expanded to this end.

### 7.1 Further work

In this paper we largely ignored the finer issues of efficiency. Our aim was to support *in-principle* efficient single-node compilation, which happens when the DKrivine machine executes trivially on a single node as a Krivine machine, and to reduce the communication overhead by sending only small (bounded-size) messages which are necessary. For example, our use of *views* of remote stack extensions avoids the need to send *pop* messages. In the future we would like to examine the possibility of *efficient* compilation on a hunch that this could be a practical programming paradigm for distributed computing. In order to do this several immediate efficiency issues must and can be addressed.

**Remote pointers** In the RPC literature it is sometimes argued that a shared or virtual address space, which is where our distributed heap of continuation stacks lives, is prohibitively expensive. However, research progress in tagged pointer representation [24] suggests that we can use pointer tags to distinguish between local and remote pointers without even having to dereference them. With such tags we would pay a very low, if any, performance penalty for the local pointers.

**Garbage collection** The execution of the Krivine Net creates garbage in the machines. Distributed garbage collection can be a serious problem [29], but we have strong reasons to believe that it can be avoided here, because the heap structures that get created are quite simple. Most importantly, there are never circular linked structures, otherwise the relations would not be well founded. This means that a simpler method, reference-counting, can be used [4]. We also know that efficient memory management is possible when compiling CBN functional programming languages to distributed architectures. The GOI compiler is purely stack-based, while the GAMC compiler uses heaps but does explicit deallocations of locations that are no longer needed.

**Shortcut forwarding** One of the most unpleasant features of the current Krivine Net approach is the excessive forwarding of data, especially on remote **RETURN**. A way to alleviate this issue might be to not create indirections when a node has a stack consisting only of a stack extension at the time of a remote invocation, meaning that the remote node could return directly to the current node's invoker. However, the implementation is

complex enough to raise non-trivial issues of correctness and therefore this falls outside the scope of this paper.

We also plan to improve the programming language by adding more expressiveness, such as parallelism, assignable state and algebraic datatypes. Some of these features have been already implemented in a compiler based on game semantics, but we hope they will be more efficient in the current setting. We also aim to enrich the type system to pay more attention to possibly unrealistic patterns of distributions. Type systems such as ML5 [35] can ensure that the interaction between local and remote resource is safe – in our approach all such interactions are *safe*, but some can be extremely inefficient and a type system can issue at least warnings against unreasonable deployments. Finally, another language-level development that could be useful is the principled development of configuration, deployment and, more generally, choreography languages.

Our dream is the eventual development of an end-to-end seamless distributed compiler for a higher-order imperative and parallel functional programming language, along the lines of the CompCert project [20]. The formalisation of the correctness of the Krivine Net, relative to the conventional Krivine machine, is the first, but technically the most demanding step.

## Acknowledgments

We acknowledge the support of the UK EPSRC and of Microsoft Research. We discussed our project with a large number of people and their feedback was invaluable in steering us in a hopefully reasonable direction: Satnam Singh, Louis Ryan, Pavan Adharapurapu and David Espinosa (Google); Guido van Rossum (Dropbox); Peter Sewell (Cambridge); Simon Peyton-Jones and Nick Benton (Microsoft Research). Lastly, we thank the anonymous reviewers for their insightful comments and criticism.

## References

[1] Krivine Nets, proofs and compiler implementation. `https://bitbucket.org/ollef/krivine-nets`. Last accessed: 3 June 2014.

[2] Amal J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In Peter Sestoft, editor, *ESOP*, volume 3924 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2006.

[3] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217 – 248, 1992. Selected Papers of the 2nd Workshop on Concurrency and Compositionality.

[4] D. I. Bevan. Distributed garbage collection using reference counting. In J. W. de Bakker, A. J. Nijman, and Philip C. Treleaven, editors, *PARLE (2)*, volume 259 of *Lecture Notes in Computer Science*, pages 176–187. Springer, 1987.

[5] Silvia Breitinger, Ulrike Klusik, Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Pena. Dream: The distributed eden abstract machine. In Chris Clack, Kevin Hammond, and Antony J. T. Davie, editors, *Implementation of Functional Languages*, volume 1467 of *Lecture Notes in Computer Science*, pages 250–269. Springer, 1997.

[6] Nadia Busi, Roberto Gorrieri, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Choreography and orchestration: A synergic approach for system design. In Boualem Benatallah, Fabio Casati, and Paolo Traverso, editors, *ICSOC*, volume 3826 of *Lecture Notes in Computer Science*, pages 228–240. Springer, 2005.

[7] Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. Designing distributed applications with mobile code paradigms. In W. Richards Adrion, Alfonso Fuggetta, Richard N. Taylor, and Anthony I. Wasserman, editors, *ICSE*, pages 22–32. ACM, 1997.

[8] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer, 2006.

[9] Ezra Cooper and Philip Wadler. The RPC calculus. In António Porto and Francisco Javier López-Fraguas, editors, *PPDP*, pages 231–242. ACM, 2009.

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[11] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *LCN*, pages 455–462. IEEE Computer Society, 2004.

[12] Olle Fredriksson. Distributed call-by-value machines. *CoRR*, abs/1401.5097, 2014.

[13] Olle Fredriksson and Dan R. Ghica. Seamless distributed computing from the geometry of interaction. In Catuscia Palamidessi and Mark Dermot Ryan, editors, *TGC*, volume 8191 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 2012.

[14] Olle Fredriksson and Dan R. Ghica. Abstract machines for game semantics, revisited. In *LICS*, pages 560–569. IEEE Computer Society, 2013.

[15] Dan R. Ghica. Applications of game semantics: From program analysis to hardware synthesis. In *LICS*, pages 17–26. IEEE Computer Society, 2009.

[16] Daniel Hirschkoff, Damien Pous, and Davide Sangiorgi. An efficient abstract machine for safe ambients. *J. Log. Algebr. Program.*, 71(2):114–149, 2007.

[17] Kohei Honda. Session types and distributed computing. In Artur Czumaj, Kurt Mehlhorn, Andrew M. Pitts, and Roger Wattenhofer, editors, *ICALP (2)*, volume 7392 of *Lecture Notes in Computer Science*, page 23. Springer, 2012.

[18] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *J. Funct. Program.*, 2(2):127–202, 1992.

[19] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007.

[20] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 42–54. ACM, 2006.

[21] Hans-Wolfgang Loidl, Fernando Rubio, Norman Scaife, Kevin Hammond, Susumu Horiguchi, Ulrike Klusik, Rita Loogen, Greg Michaelson, Ricardo Pena, Steffen Priebe, Álvaro J. Rebón Portillo, and Philip W. Trinder. Comparing Parallel Functional Languages: Programming and Performance. *Higher-Order and Symbolic Computation*, 16(3):203–251, 2003.

[22] Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña-Marí. Parallel functional programming in eden. *J. Funct. Program.*, 15(3):431–475, 2005.

[23] Ian Mackie. The geometry of interaction machine. In Ron K. Cytron and Peter Lee, editors, *POPL*, pages 198–208. ACM Press, 1995.

[24] Simon Marlow, Alexey Rodriguez Yakushev, and Simon L. Peyton Jones. Faster laziness using dynamic pointer tagging. In Ralf Hinze and Norman Ramsey, editors, *ICFP*, pages 277–288. ACM, 2007.

[25] Derek Gordon Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: A universal execution engine for distributed data-flow computing. In David G. Andersen and Sylvia Ratnasamy, editors, *NSDI*. USENIX Association, 2010.

[26] Kensuke Narita and Shin-ya Nishizaki. A parallel abstract machine for the RPC calculus. In *Informatics Engineering and Information Science*, pages 320–332. Springer, 2011.

[27] Ulf Norell. Dependently typed programming in agda. In Andrew Kennedy and Amal Ahmed, editors, *TLDI*, pages 1–2. ACM, 2009.

[28] Atsushi Ohori and Kazuhiko Kato. Semantics for communication primitives in an polymorphic language. In Mary S. Van Deusen and Bernard Lang, editors, *POPL*, pages 99–112. ACM Press, 1993.

[29] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In Henry G. Baker, editor, *IWMM*, volume 986 of *Lecture Notes in Computer Science*, pages 211–249. Springer, 1995.

[30] Sigram Schindler. Standards and protocols: Distributed abstract machine. *Comput. Commun.*, 3(5):208–220, October 1980.

[31] James W. Stamos and David K. Gifford. Remote evaluation. *ACM Trans. Program. Lang. Syst.*, 12(4):537–565, 1990.

[32] Philip W. Trinder, Hans-Wolfgang Loidl, and Robert F. Pointon. Parallel and Distributed Haskells. *J. Funct. Program.*, 12(4&5):469–510, 2002.

[33] David Turner et al. *The polymorphic pi-calculus: Theory and implementation*. PhD thesis, University of Edinburgh. College of Science and Engineering. School of Informatics., 1996.

[34] Vasco Thudichum Vasconcelos, Simon J. Gay, and António Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006.

[35] Tom Murphy VII, Karl Crary, and Robert Harper. Type-safe distributed programming with ML5. In Gilles Barthe and Cédric Fournet, editors, *TGC*, volume 4912 of *Lecture Notes in Computer Science*, pages 108–123. Springer, 2007.