Journal of Universal Computer Science, vol. 9, no. 8 (2003), 776-794
submitted: 24/2/03, accepted: 30/5/03, appeared: 28/8/03 © *J.UCS*

$\text{Haskell}_{\#}$: Parallel Programming Made Simple and Efficient

Francisco Heron de Carvalho Junior
(Centro de Informática
Universidade Federal de Pernambuco, Recife, Brazil
fhcj@cin.ufpe.br)

Rafael Dueire Lins
(Departamento de Eletrônica e Sistemas
Universidade Federal de Pernambuco, Recife, Brazil
rdl@ee.ufpe.br)

Abstract: This paper presents the final result of the designing of a new specification for the $\text{Haskell}_{\#}$ Language, including new features to increase its expressiveness, but without losing either efficiency or obedience to its original premisses.
Key Words: Fuctional Programming, Petri Nets, Parallelism, Languages.
Category: C.2.4, D.1.1, D.1.3, D.2.2, D.3.2, D.3.3

1 Introduction

$\text{Haskell}_{\#}$ ("Haskell hash") is an explicit parallel distributed language, defined on top of Haskell, that has evolved since 1998 [Lima and Lins 1998]. The premisses that guide its design attempt to make parallel programming a task reachable for most programmers, without having to pay for loss of efficiency of parallel programs when running over distributed architectures, such as *clusters* [Baker et al. 1999]. Below, we discuss them:

- Ideally, parallelism would be implicit, freeing programmers from the burden of control communication and synchronization concerns. However, practice has demonstrated that efficient *implicit parallelism* is hard to obtain in its general case, due to the high complexity of the configuration tasks required to parallelize sequential programs, such as partitioning, allocation, granularity control, and so on. To parallelize efficiently, compilers should take into account many aspects of the intrinsic features of the target architecture and the application. This is not always easy to model. Even in functional languages, where parallelism is easier to detect, the results obtained are very poor, if compared to explicit approaches [Loidl et al. 2000];
- Dynamic control of parallelism, involving tasks such as process migration for load balancing processors and on demand creation of processes, generates a

high overhead in the run-time of parallel programs, increasing proportionally to the communication latency amongst processors of the target distributed architecture;

- In parallel programs, the interleaving of the primitives for control of parallelism and the computation code makes the analysis of formal properties very hard. This way, it is impossible to abstract process interaction from computation;
- The mixture of computation and parallelism control code also makes programming difficult, requiring skilled and well trained parallel programmers, increasing the costs of the development of complex parallel applications;
- The lack of an agreed model for parallel programming lessens portability and the chances for the existence of systematic parallel development methods and tools. The intimate relationship between existing programming models and parallel architectures serves to explain the diversity of the former [Skillicorn and Talia 1998].

Haskell$_{\#}$ was designed based on the above assumptions. Thus, it now supports the following features:

- **Explicit and static parallelism configuration.** This minimizes overheads in the execution time of parallel programs, assuming that the programmer is the most capable specialist to perform the configuration of parallel tasks efficiently, due to his/her knowledge about target architecture and the application and the non-existence of efficient algorithms to perform optimally configuration tasks in their general instances automatically;
- **Efficient and simple implementation.** Haskell$_{\#}$ only needs to "glue" a Haskell sequential compiler to a message passing library. GHC and MPI, respectively, have been used in this work. The use of an efficient sequential Haskell compiler is important, assuming that Haskell$_{\#}$ style of programming encourages coarse grained parallelism, where most of time is expended in performing sequential computations;
- **Abstraction of parallelism from computation.** In essence, Haskell$_{\#}$ encompasses a coordination language [Gelernter and Carriero 1992] [Carvalho Jr. et al. 2002a]. Computations (Functional Processes) are described in Haskell, while coordination (parallelism configuration) is described by means of HCL (Haskell$_{\#}$ Configuration Language). HCL is syntactically orthogonal to Haskell. No extensions or libraries are necessary in Haskell for gluing processes to coordination medium, described in HCL. This characteristic induces a *process hierarchy*, with several benefits, such as a higher

degree of modularity, in many aspects of programming, and simplication of programming tasks;

- **Equivalence to Petri Nets.** HCL descriptions can be translated into (labeled) Petri nets[Petri 1966] that describe the process interaction and communication behaviour of parallel programs, making possible effective formal analysis of their properties[Carvalho Jr. et al. 2002b];
- **Hierarchical Compositional Programming.** Compositional programming is an important technique to increase the modularity of programming languages, most specially in parallel ones [Foster 1985]. Hierarchical composition is an important feature supported by modern configuration languages in distributed systems[Krammer 1994, Magee et al. 1995]. It adds to Haskell$_{\#}$ new abstraction mechanisms for describing complex network topologies in a simple way;
- **Partial Topological Skeletons.** Skeletons are an important programming technique developed by Murray Cole[Cole 1989]. They allow to orthogonalise the description of an algorithm from its efficient implementation in a specific architecture. General reusable patterns of process interaction found in concurrent programs define a skeleton. Haskell$_{\#}$ partial topological skeletons make it easier to describe complex network topologies, giving support for *nesting* and *overlapping* operations to allow composition of skeletons from existing ones[Carvalho Jr. 2003].

Three other sections compose this paper. Section 2 briefly explains the evolution of Haskell$_{\#}$ since its original conception. Section 3 describes the structure of Haskell$_{\#}$ programs, and also presents some representative examples. Finally, Section 4 draws some conclusions and lines for current and further works in Haskell$_{\#}$.

2 The Evolution of Haskell$_{\#}$

Since the publication of its original design in 1998 [Lima and Lins 1998], three versions of the Haskell$_{\#}$ language have appeared. Each version has tried to improve upon the previous one in meeting the targets described in Section 1 of this paper.

In the first Haskell$_{\#}$ version [Lima et al. 1999], funcional processes communicated by making explicit calls to message passing primitives defined on top of the Haskell IO monad. HCL was used to define the communication topology of processes, connected via OCCAM-style channels[Inmos, 1984]. The use of explicit message passing primitives extending Haskell allows a mixture of the primitives for the synchronisation of parallelism with computational code, breaking down

Haskell$_{\#}$ principle of *process hierarchy*. This makes nearly impossible the analysis of formal properties of programs with Petri nets or any other formalism.

The first revision of Haskell$_{\#}$ produced a new version, where explicit message passing primitives were abolished in favor of process hierarchy. Communication input and output ports of functional processes became mapped onto arguments and elements of the tuple returned by their *main* functions, respectively. Ports are connected to define communication channels, as in the first version. Functional Processes were strict, thus parameters are transmitted in their normal form as values. A process is executed by performing the following tasks in sequence: read input ports in the order in which they were declared in the HCL program; call the *main* function, passing as arguments the values received by input ports, in the order they were received; send each element of the tuple returned by function *main* to the output ports in the order they were declared.

Not allowing functional processes to perform communication interleaved with computation made it difficult to express important concurrent patterns of process interaction used in some parallel applications, such as *systolic* and *pipe-line* computations. These applications require that processes exchange information while they compute, or keep the state of computation between communication operations during computation. However, this version made possible to define the first translation of Haskell$_{\#}$ programs into Petri nets and to analyse the communication behaviour of some applications, such as an ABS (*Anti-Lock Braking System*) control unit [Lima and Lins 2000]. This was very important to demonstrate the potential of Haskell$_{\#}$ approach for parallel programming.

The lastest version of Haskell$_{\#}$, described in this paper, was developed to reconcile maximal concurrency expressiveness with process hierarchy and also to provide a higher abstraction programming mechanism without loss of efficiency. Special attention has been dedicated to the translation into Petri nets. Lazy stream communication was introduced to allow processes to interleave communication and computation operations in a transparent way. From the functional module point of view, lazy streams are Haskell lazy lists whose elements can be read or written on demand. Ports now can be read on demand. The support for skeletons and hierarchical compositional programming allow for a more abstract and higher-level style of programming. This version of Haskell$_{\#}$ is described in the following sections.

3 The Structure of Haskell$_{\#}$ Programs

This section describes the structure of Haskell$_{\#}$ programs, illustrating syntactic aspects by means of examples. The main example used here is a Haskell$_{\#}$ implementation of a well known systolic algorithm for matrix multiplication, described in [Manber 1999]. This example allows to demonstrate some important aspects

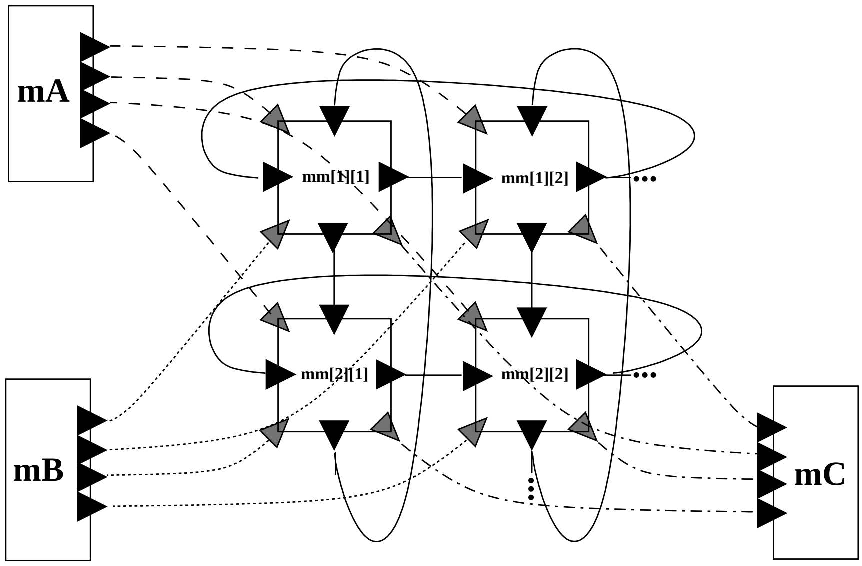

Figure 1: Matrix Multiplication Topology

of Haskell$_{\#}$ programming, such as the use of skeletons, hierarchical composition, etc.

The implementation of the matrix multiplication follows the network structure presented on Figure 1. Two processes, mA and mB, distribute matrices, named A and B, respectively, element by element, amongst a collection of processes organized in a square grid. These processes are responsible for calculating the resulting matrix C, the multiplication of the matrices A and B. The indexes that identify each process indicate its position (line and column) in the matrix of processes. The process mC collects matrix C from the processes in the grid. The Haskell$_{\#}$ skeletons *FARM* and *TORUS* [Carvalho Jr. 2003] are used to compose the network topology of processes in the matrix multiplication program.

Figures 2 and 3 present the HCL code for the Matrix Multiplication component and for the functional module Haskell code of the cooperating processes in the grid, respectively. The declarations in HCL code will be explained in the rest of the paper.

```
– In File matrix_multiplication.hcl
component MatrixMultiplication<N> with

index i, j range [1,N]

use module MatrixMultiplication.MatMult
use module MatrixMultiplication.ReadMatrix
use module MatrixMultiplication.ShowMatrix

use configuration Skeletons.Common.FARM
use configuration Skeletons.Common.TORUS

– interface for processes that compose the systolic mesh
interface IMatMult (aij, bij::Array (Int,Int) Float; l*, t*::Float)
                  → (r*, b*::Float, cij::Array (Int,Int) Float)
          behaving as seq { par {aij?; bij?};
                            repeat alt {counter < N → seq {l?; t?; r!; b!}};
                            cij! }

– process units that configure functionality of the application
unit mA as ReadMatrix N → out # Distributor () → out
unit mB as ReadMatrix N → out # Distributor () → out
unit mC as ShowMatrix in → () # Collector in → ()

– cluster units (skeletons) that configure topological organization of processes
unit farmA as FARM<N*N>
unit farmB as FARM<N*N>
unit mmgrid as TORUS<N>

– overlapping skeletons to configure final topology
[/ unify farmA.worker[(i+1)*N + j] # a → c,
         farmB.worker[(i+1)*N + j] # b → c,
         mmgrid.meshcell[i][j] # (l,u) → (r,d)
              to matmult[i][j] as MatMult (N,a,b,l,u) → (r,d,c)
                                # IMatMult (a,b,l,u) → (r,d,c) /]

unify mmfarmA.collector # c → (),
      mmfarmB.collector # c → () to showmatrix # c → ()

– mapping processes onto the topology above defined (overlapping of the skeletons)
assign mA to mmfarmA.distributor () → (divide_matrix out)
assign mB to mmfarmB.distributor () → (divide_matrix out)
assign mC to showmatrix (combine_matrix in) → ()
```

Figure 2: HCL code for a matrix multiplication on a circular mesh

3.1 Units: The Building Blocks of HCL Configurations

Haskell$_{\#}$ programs are essentially described by HCL configurations. The *main unit* defines the computation of the program. An *interface* and a *component* should be associated with each unit and are detailed as follows.

```
– In File matmul.hs
module MatMult(main) where

type MatrixElem = Array (Int,Int) Double

main :: Int→MatrixElem→MatrixElem→[Int]→[Int]→ IO (MatrixElem,[Int],[Int])
main n aij bij as bs = return cij
      where
              ((i,j),aij') = (head.assocs) aij
              ((_,_),bij') = (head.assocs) bij
              cij' = (matmul (aij'*bij') n aij bij as bs)
              cij = array (1,1) [((i,j),cij')]

matmul :: Float→Int→Float→Float→[Float]→[Float] → (Float,[Float],[Float])
matmul c 1 _ _ _ _ = (c,[],[])
matmul c n aij bij (a:as) (b:bs) = (cij,aij:as',bij:bs')
              where
                     (cij,as',bs') = matmul (c+a*b) (n-1) a b as bs
```

Figure 3: The MatMult Functional Module

3.1.1 Interfaces: *Describing How Units Interact*

An interface defines the set of input and output ports of the unit for communicating with each other, as well as the order in which they are *activated* (communication behaviour). The order is defined by an expression written in an OCCAM-like [Inmos, 1984] language, designed in such way that its expressive power is equivalent to expressive power of labelled Petri nets. The HCL piece of code below defines an interface:

```
interface SystolicMeshCell (left*, above*::t) → (right*, below*::t)
          behaving as Pipe # left → right
                    as Pipe # above → below
                    as: repeat alt {
                              right! → below!; below! → right!;
                              common: par { left?; above? }
                       }
```

The interface *SystolicMeshCell* has two input ports, named *left* and *above*, and two output ports, named *right* and *below*. All ports have some unknown type (polymorphism). The * after the name of each port means that the port transmits a *stream* of values of type *t*. The **behaving as** clause introduces *behaviour constraints*. In the example, *SystolicMeshCell* has three behaviour constraints. The first one says that a systolic mesh cell must behave like a pipe-line stage when considering only ports *left* and *right*, which make the role of the *input* and the *output* ports of a pipe-line stage, respectively. The second statement works similarly for ports *above* and *below*. The last declaration specifies the *communication behaviour* of the units that use this interface. A language based on 0-counter synchronised regular expressions is used, a formalism equivalent to

Petri nets. The compiler has to ensure the compatibility of the specified communication behaviour with the other behaviour constraints, producing an error message whenever necessary. An interface may have many behaviour restrictions (like the first two in the example above), but at most one could describe a *communication behaviour*. If the programmer does not specify the communication behaviour, the compiler generates the *weakest communication behaviour* by default. This behaviour must satisfy any the other existing behaviour constraint. The generation algorithm is not detailed in this paper.

The combinators used to build communication behavior descriptions are described below:

- **seq**: Specifies a strict order for a collection of communication actions (sequential composition) ;
- **par**: Denotes interleaved execution of a collection of communication actions (concurrent composition) ;
- **alt**: Stands for alternative execution amongst a collection of actions (alternative composition). Each actions is guarded by a watchdog port. The choice of the action to be performed follows the semantics of CSP;
- **repeat alt**: Specifies repeated execution of a collection of alternative actions. Here, watchdog ports must be streams. The termination condition is reached when all streams communicated by all watchdog ports receives the EOS (end of stream) message. A special type of guard can be used to control the number of iterations. Its form is **counter** $< N$, where N is an HCL numeric expression. Also, the **skip** guard is used to model a guard that is always active (default guard). It can be used to model infinite repetition.
- **p?**: Indicates activation of input port p, trying to receive a message its communication pair;
- **p!**: Denotes activation of output port p, trying to send a message to its communication pair;
- **wait** *sem*: Corresponds to the *wait* primitive in a counter semaphore *sem*. The action is delayed until a sufficient number of **signal** *sem* actions are performed;
- **signal** *sem*: Corresponds to the *signal* primitive in a counter semaphore *sem*;

3.1.2 Components: *Describing What Units Compute*

A component defines the computation performed by a unit. It can be *simple* or *compound*. Simple components are Haskell programs, called functional modules, that are the primitive computational entities of Haskell$_\#$ programs. Units instantiated from simple components are called *processes*. Compound components are HCL configurations, defining a network of cooperating units. Units instantiated from compound components are called *clusters*. The use of clusters provide a higher degree of modularity and potential for reuse of code at coordination level of Haskell$_\#$ programming, by means of hierarchical compositional programming and skeleton programming.

Programs in Figure 2 and 3 illustrate a compound component declaration (HCL configuration) and a functional module, respectively. In the compound component *MatrixMultiplication*, the parameter `N` defines the dimension of the matrices. The declarations that follow configure the functional process network, whose diagram is shown in Figure 1. In the functional module *MatrixMult*, one may observe that there are no extensions or special libraries introduced into the Haskell code, due to the total orthogonality between HCL and Haskell.

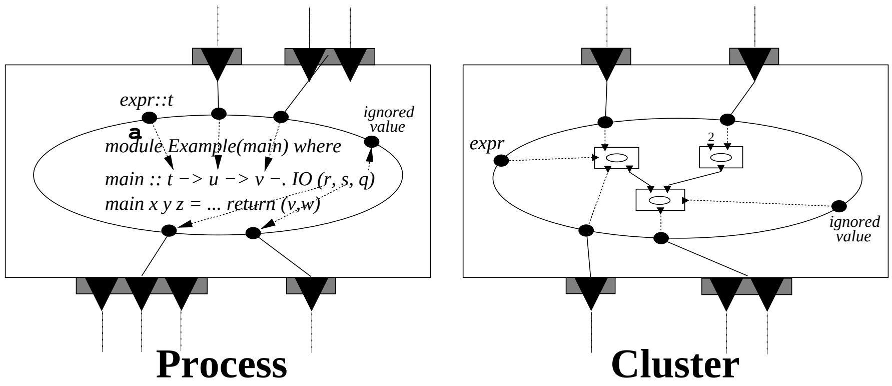

Figure 4: Entry and Exit Points

Components may have entry and exit points. For functional modules (simple components), entry points are the arguments of the function *main*, while exit points are the elements of the tuple returned in the resulting IO action. For HCL configurations (compound components), entry and exit points are linked, respectively, to input and output ports that are not connected to other ports. Entry and exit points are illustrated in Figure 4. In the compound component declared in Figure 10, one entry point and one exit point are declared, respectively named *in* and *out*. The bind declaration at the end of the HCL configuration declares that the input and output ports are bound to the entry and exit points.

An injective mapping from entry and exit points of the component onto input and output ports, respectively, must be defined. Values must be provided explicitly to the entry points that are not associated with an input port. The value produced by exit points not connected to an output port is ignored. A wire function should be specified to transform values between entry and exit points and input and output ports. If not specified, the identity function is assumed. The wire function maps a value of type t to a type u.

In the hierarchical view of any Haskell$_\#$ program, instantiated in Figure 5, the units at the leaves are processes, while internal units are clusters. The units that have the same parent node belong to the same HCL configuration.

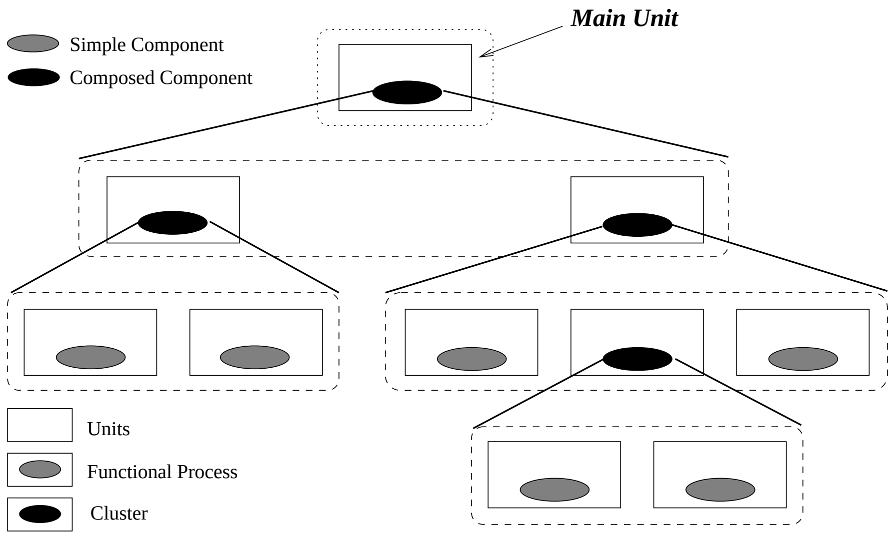

Figure 5: Unit Hierarchy in a Haskell$_\#$ Program

3.1.3 Repetitive and Non-Repetitive Units

There are two kinds of processes: *non-repetitive* and *repetitive*. The former ones reach final state after evaluating the function *main*, while the latter will execute forever, by calling repeated and sequentially their function *main*. A cluster is said to be repetitive if all units that compose it are repetitive. The compiler generates an error message if the programmer attempts to declare as non-repetitive a cluster where all composing unit are repetitive. A Haskell$_\#$ program that has as main unit a repetitive unit is said to be repetitive. Repetitive applications are very useful to implement iterated reactive applications, programs that neither reach a final state nor store state information between iterations. In general, those are reactive systems such as some operating or control systems. An example of such a declaration of a repetitive unit is shown below:

unit * random **as** Random # seed → number

The * symbol after the keyword denotes that the unit *random* is repetitive. Each time the unit (process) random is activated, a random number is generated according to a *seed* value. Notice that to obtain a different random value, it is necessary to provide different seeds at each unit activation.

3.1.4 How to Declare a Unit

The HCL declaration below declares `N` units in a single line, using indexed notation supported by HCL:

index i, j **range** [1,`N`]
⋮
[/ **unit** grid_cols[i] **as** PIPE-LINE<`N`> in → out # *Pipe* in → out /]

The **index** declaration specifies two indexes, i and j, whose range is configured in **range** clause, from 1 to `N`, where `N` is a configuration parameter. The tokens [/ and /] delimit the scope for variation indexes used in their context. In the example, notice that i is the index that is in the scope of the delimiters. `N` units are declared, named *grid_cols*, which are indexed from 1 to `N`. The *grid_cols* units are instances of the compound component *PIPE-LINE*, forming clusters that represent a topological structure. Their interface is *Pipe*, defining one input port, named *in* and one output port, named *out*. The entry and exit points of *PIPE-LINE* component are mapped onto input and output interface ports of the unit by matching identifiers. In the declaration of a toroidal mesh skeleton (Figure 8), the ports *in* and *out* of *grid_cols* clusters will be connected to define circular pipe-lines.

3.1.5 Replication of Ports in Unit Declarations

In a unit declaration, after mapping component entry and exit points to interface ports, it is still possible to replicate an interface port, forming a group of ports. Groups of ports are associated with either an entry or exit point. The default wire function for groups of ports is *choice*, that chooses one of the ports in the group to perform communication. The wire function for input ports must have one of the following types:

wire_in :: $Int \rightarrow (Int \rightarrow t) \rightarrow u$

wire_in_io :: $Int \rightarrow (Int \rightarrow IO\ t) \rightarrow IO\ u$

The programmer can choose between IO or non-IO version. The first argument is the number of ports in the group, while the second one is a function that receives an integer, that indicates the index of a port, and returns the value, of type t, received by that port. The returned value of type u is passed to the associated entry point.

The wire function for output ports can have one of the following types:

wire_out :: $Int \rightarrow u \rightarrow (Int \rightarrow t)$

wire_out_io:: $Int \rightarrow u \rightarrow (Int \rightarrow IO\ t)$

The first argument specifies the number of ports in the group, while the second one is the value produced in the exit point of the component. A function must be returned, which receives an integer that indicates the index of an output port and returns the value sent through it.

Figure 2 shows how groups of ports are declared in interface specification of units mA, mB and mC.

3.2 Communication Channels

Haskell$_{\#}$ communication channels are used to connect ports of opposite directions from two units, forming the network topology of a HCL configuration. They are *synchronous*, *point-to-point*, *unidirectional* and *typed*. In this new Haskell$_{\#}$ version, *bounded buffers* are introduced to allow a weak form of asynchronous communication. The declaration below illustrates the syntax of a channel declaration in HCL :

index i, j **range** [1,N]
⋮
[/ **connect** * grid_cols[i].pipe[N]→out **to** grid_cols[i].pipe[1]←in **buffered** 10 /]

In the example above, N channels are declared. Let v be an index value (value of i), the channel links the port *pipe[N]* of unit *grid_cols[v]* to the port *pipe[1]* of the same unit. The * after the **connect** combinator indicates that the channel transmits a stream. The ports connected by a stream channel must be declared to be streams too, or the compiler will generate an error message. The clause **buffered** is optional and establishes a buffered channel. The buffer size is configured as an integer value after **buffered** clause. In the example, its value is 10. If omitted, the buffer size is assumed to be limited to the amount of memory available.

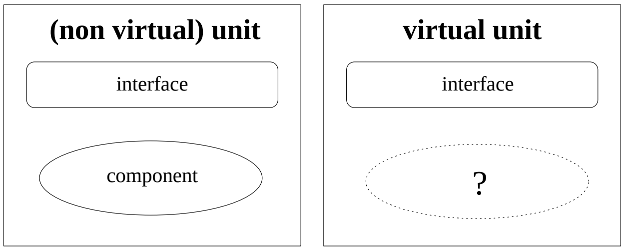

Figure 6: Virtual and non-virtual

3.3 Virtual Units: *Towards Skeleton Programming*

It is possible to define a unit without a component, but an interface is always required. When an interface is not provided, a default one is implicitly assumed. A unit for which a component does not exist is called a *virtual unit* (Figure 6).

Virtual units are essentially templates from which non-virtual units can be defined. At any moment, a non-virtual unit can be *assigned* to a virtual unit, occupying its place and assuming its role in the network of units. For that, there must be a behaviour constraint that associates the interface of the assigned unit with the interface of the virtual unit. Behaviour constraints were discussed in Section 3.1.1.

The syntax for declaration of virtual units is presented on Figures 7 and 8. It is similar to a non-virtual unit declaration, but the **virtual** keyword is used before the **unit** keyword. The clause **as**, used to define the component, is not necessary. An example of use of assignment operation is presented on Figure 2. The process mA is assigned to the virtual unit *distributor* of the cluster *mmFarmA*. In the declaration of process mA. It is configured to be the interface *Distributor*, the same of the virtual unit *distributor*, allowing the compiler to perform the appropriate assignment.

Virtual units are at the heart of the Haskell$_{\#}$ support for skeleton programming, an important programming technique introduced by Cole[Cole 1989]. In his seminal work, Cole defined the notion of algorithm skeletons to capture common patterns found in large classes of programs. The motivation for skeletons is the possibility to use them to increase reuse and modularity of code and to allow to abstract implementation concerns from specification of algorithms, without compromising their performance. Each skeleton of an algorithm could be implemented for each target architecture in an optimized way. An important point to notice is that, in the research developed about skeletons in the subsequent years after Cole introduce the idea, special attention was deserved to skeletons that captured patterns of concurrent and parallel interaction, as a way to make parallel programming more abstract. As a consequence, several parallel languages now support skeletons[Skillicorn and Talia 1998].

Haskell$_{\#}$ introduces the notion of *partial topological skeleton*, which captures common patterns of topology and communication behaviour. A skeleton in Haskell$_{\#}$ is essentially a component with at least one virtual unit. If all of them are virtual, then this component is called a *total skeleton*, a widely adopted assumption. However, in Haskell$_{\#}$ skeletons, some of the computational entities (units) can be completely defined, having a component and an interface. Therefore, Haskell$_{\#}$ skeletons are partial. The compound components in Figures 7 (*FARM*), 8 (*MESH*), 9 (*TORUS*), and 10 (*PIPE-LINE*) are examples of common total *skeletons*, once they define virtual units in their compositions.

Besides providing a higher degree of modularity and potential for reuse of code at coordination programming level, another important use of skeletons is to provide to the HCL compiler explicit information about the topology of the functional processes. Depending on the skeleton used to organise a certain group of processes, the compiler can use special rules to generate more efficient parallel code, more appropriate to the execution environment. For example, Haskell$_{\#}$ provides skeletons that abstract collective communication primitives of MPI, allowing the compiler to make more efficient use of them. Another possibility that has been investigated is to allow programmers to specify, in a meta-language, how MPI code is generated for a given skeleton.

```
– In file farm.hcl
component FARM<N> with

export interface Distributor, Worker, Collector

interface Distributor () → out behaving as: repeat alt {out! → skip}
interface Worker      in → out behaving as: repeat alt {in? → skip}
interface Collector   in → ()  behaving as: repeat alt {in? → skip}

virtual unit distributor # Distributor () → out
virtual unit worker      # Worker      in → out
virtual unit collector   # Collector   in → ()

connect distributor→out to worker←in
connect worker→out      to collector←in

replicate N worker
```

Figure 7: FARM Skeleton

```
– In file mesh.hcl
component MESH<N> # # ([/ l[i] /], [/ u[i] /]) → ([/ r[i] /], [/ d[i] /]) with

use configuration Skeletons.Common.PIPE-LINE
export interface SystolicMesh

index i, j range [1,@N]

interface SystolicMeshCell (left*, above*::t) → (right*, below*::t)
          behaving as Pipe # left → right
                   as Pipe # above → below
                   as: repeat alt {
                             right! → below!; below! → right!;
                             common: par { left?; above? }
                      }

[/unit grid_cols[i] as PIPE-LINE<N> # in → out/]
[/unit grid_rows[i] as PIPE-LINE<N> # in → out/]

[/ bind grid_cols[i].pipe[1]→out to u[i] /]
[/ bind grid_cols[i].pipe[N]→out to d[i] /]
[/ bind grid_rows[i].pipe[1]→out to l[i] /]
[/ bind grid_rows[i].pipe[N]→out to r[i] /]

[/unify grid_cols[i].pipe[j] # l → r, grid_rows[j].pipe[i] # t → b
       to cell[i][j] # SystolicMeshCell (l,t) → (r,b)/]
```

Figure 8: MESH Skeleton

```
– In file torus.hcl
component TORUS<N> with

index i range [1..N]

use configuration Skeletons.Common.MESH

unit torus as MESH # ([/ l[i] /], [/ u[i] /]) → ([/ r[i] /], [/ d[i] /])

[/ connect torus→r[i] to torus→l[i] /]
[/ connect torus→d[i] to torus→u[i] /]
```

Figure 9: TORUS Skeleton

3.3.1 Unification and Factorization of Virtual Units

Two operations can be performed with virtual units: *unification* and *factorization.* The former allows for programmers to compose new virtual units from a set of virtual units, by composing them. The interface of the new virtual unit assumes behavioural constraints that relate it to the interfaces of the original units. This means that the new unit must preserve the behaviour of the original

```
– In File pipe.hcl
component PIPE-LINE&lt;N&gt; in → out with

index i range [1,N-1]
index j range [1,N]

export interface Pipe in*::[t] → out*::[u]
                 behaving as: repeat alt {in? → out!}

[/virtual unit pipe[j] # Pipe in → out;/]

[/channel link[i] connect * pipe[i]→out to pipe[i+1]←in/]

bind pipe[1]←in to in
bind pipe[N]→out to out
```

Figure 10: PIPE Skeleton

units in the network. Factorisation is the dual of unification, allowing for a unit to be divided into two or more units, preserving the behaviour of the original units, by applying the restrictions of their interface.

An example of unification is presented in the HCL code for the MESH skeleton component on Figure 8. The units that compose the N *grid_cols* pipe-line units and N *grid_rows* pipe-line units are unified in a mesh. This is an example of **overlapping** of skeletons, which can be used to compose a new skeleton as demonstrated in the presented example. Another way to compose new skeletons from existing ones is **nesting**, where a skeleton unit is assigned to a virtual unit of another skeleton.

In general, in Haskell$_{\#}$ programming, it is convenient to make use of skeletons to define topology of processes, without making use of the **connect** clause. The skeletons are overlapped using unification and factorisation operations (described below), forming a network skeleton of the program. Then, processes are assigned to the resulting virtual units. This technique is used to build the matrix multiplication configuration on Figure 2.

3.4 Replication of Units: *Configuring Distributed Data Parallel Computations*

An operation is defined over any kind of unit: *replication*. It allows the construction of many copies of a unit in the network. Amongst other uses, replication is suitable to define distributed data-parallel computations. An example of replication is shown in *FARM* skeleton declaration on Figure 7. A unit named *worker* is defined and then replicated to form the set of farm worker processes, indexed from 1 to *N*. Special attention must be given to the ports connected to the replicated process. They must be also replicated, in such way that now there is a copy of the original port for each unit derived from replication. Notice that, in the example, the resulting groups of ports of units *distributor* and *collector* have no wire function defined. When assigning non-virtual units to FARM virtual ones, the programmer should define these functions, configuring the FARM according to his needs. An example of this approach can be seen in the matrix multiplication HCL code 2, where wire functions *dist_matrix* and *combine_Matrix* are configured to be the required wire function of FARM's $farmA$, $farmB$, and $farmC$, after the assignment of mA, mB and mC to their virtual units.

3.5 Startup Modules: *Putting Things to Work*

Startup modules define the *main unit* of the application, giving, whenever necessary, values to be passed onto its entry ports. If the main unit has output ports, their values produced at the end of computation can be given to an *exit* Haskell function, which performs some final computation with the yielded values. The startup module for the matrix multiplication program is presented on Figure 11. The component associated with the *main* unit is called *application component.*

```
application MatrixMultiplication with
use configuration MatrixMultiplication.MatrixMultiplication
unit main as MatrixMultiplication<4>
```

Figure 11: Matrix Multiplication Startup Module

The example below demonstrates the full functionality that could be presented in a startup module.

```
application StartupDemonstration with
{# fat 0 = 1
   fat n = n * fat (n-1)
#}
use OneComponente in 'one_component.hcl'
unit main as OneComponent<"Hello World!!!"> ([1..],fat 10) → (a::Int;_;b::Int)
{# exit :: Int → Int → IO()
   exit x y = print x >>
              print y
#}
```

The expressions [1..] and $fat10$ produce values that are passed to the entry points of the component. The underscore at the second output port declaration of the main unit says that the value of that port may be ignored. The values passed as arguments to the *exit* function at the end of the computation are that produced by output ports a and b, in the order that they appear. "Hello World!!!" is a string parameter given to the instance of the component *OneComponent*.

3.6 Libraries: *Enhancing Reuse Capabilities*

In libraries, programmers can put HCL declarations that can be imported in the HCL configuration that define distinct components, imposing a discipline for reuse of code. An example of library is presented below:

library LibraryExample(*Send*,*Receive*) **with**

interface *Send* () $\rightarrow$ out*
interface *Receive* in* $\rightarrow$ ()

The library above declares and exports two interfaces, named *Send* and *Receive*. They can be imported in a HCL configuration by using the following declaration:

import LibraryExample(**interface** *Send*, **interface** *Receive*)

4 Conclusions and Lines for Further Works

This paper describes the last efforts on the design of Haskell$_{\#}$, a language targeted at making parallel programming more abstract and modular, with concerns on efficiency and formal analysis of programs. An integrated environment development, simulation, and formal analysis of parallel programs, based on Haskell$_{\#}$, is currently being developed, in JAVA. This environment, named VHT (*Visual Haskell$_{\#}$ Tool*) should be able to manage the development of high-performance and complex applications, taking into account modularity and reuse of code, making use of skeletons and the composition of programming components. Because Haskell$_{\#}$ covers several hot topics in advanced parallel programming in a unified and simple way, VHT may be also used for educational purposes in graduate or undergraduate courses. A simulator for Haskell$_{\#}$ programs, based on network simulator tools[Fall and Varadhan 2002], and a Petri net based analysis tool are under development. VHT should offer integration to PEP[Best E. et al. 1997] and INA[Roch and Starke 1999], for the analysis of formal properties of Petri nets, giving support for automatic translation of Haskell$_{\#}$ programs into that formalism.

Acknowledgements

The authors are grateful to professor Simon Thompson careful reading and comments on this paper.

References

[Baker et al. 1999] Baker, M., Buyya, R., and Hyde, D. (1999). Cluster Computing: A High Performance Contender. *Communications of the ACM*, 42(7):79–83.

[Best E. et al. 1997] Best E., Esparza J., Grahlmann B., Melzer S., Rmer S., and Wallner F. (1997). The PEP Verification System. In *Workshop on Formal Design of Safety Critical Embedded Systems (FEmSys'97)*.

[Carvalho Jr. et al. 2002a] Carvalho Jr., F., Lima, R., and Lins, R. (2002a). Coordinating Functional Processes with Haskell$_{\#}$. In ACM Press, editor, *ACM Symposium on Applied Computing, Special Tracking on Coordination Languages, Models and Applications*, pages 393–400.

[Carvalho Jr. et al. 2002b] Carvalho Jr., F., Lins, R., and Lima, R. (2002b). Translating Haskell$_{\#}$ Programs into Petri Nets. In Faculdade de Engenharia, Universidade do Porto, editor, *Proceedings of VECPAR'2002*.

[Carvalho Jr. 2003] Carvalho Jr., F. H. (2003). Topological Skeletons in Haskell$_{\#}$. In *International Parallel and Distributed Symposium (IPDPS 2003)*. IEEE Press. *(april 2003)*.

[Cole 1989] Cole, M. (1989). *Algorithm Skeletons: Structured Management of Paralell Computation*. Pitman.

[Fall and Varadhan 2002] Fall, K. and Varadhan, K. (2002). The NS Manual (formely NS Notes and Documentation). Technical report, The VINT Project, A Collaboration between researchers at UC Berkeley, LBL, USC/ISI, and Xerox PARC.

[Foster 1985] Foster, I. (1985). Compositional Parallel Programming Languages. *ACM Transactions on Programming Languages and Systems*, 18(4):454–476.

[Gelernter and Carriero 1992] Gelernter, D. and Carriero, N. (1992). Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107.

[Inmos, 1984] Inmos (1984). Occam Programming Manual. *Prentice-Hall, C.A.R. Hoare Series Editor*.

[Krammer 1994] Krammer, J. (1994). Distributed Software Engineering. In *Proceedings of 16th International Conference on Software Engineering (ICSE)*. IEEE Computer Society Press.

[Lima et al. 1999] Lima, R. M. F., Carvalho Jr., F. H., and Lins, R. D. (1999). *Haskell$_{\#}$*: A Message Passing Extension to Haskell. *CLAPF'99 - 3rd Latin American Conference on Functional Programming*, pages 93–108.

[Lima and Lins 1998] Lima, R. M. F. and Lins, R. D. (1998). *Haskell$_{\#}$*: A Functional Language with Explicit Parallelism. *LNCS VECPAR'98 - International Meeting on Vector and Parallel Processing*, pages 1–11.

[Lima and Lins 2000] Lima, R. M. F. and Lins, R. D. (2000). Translating HCL Programs into Petri Nets. In *Proceedings of the 14th Brazilian Symposium on Software Engineering*.

[Loidl et al. 2000] Loidl, H. W., Klusik, U., Hammond, K., Loogen, R., and Trinder, P. W. (2000). GpH and Eden: Comparing two Parallel functional Languages on a Beowulf Cluster. In *2nd Scottish Functional Programming Workshop*.

[Magee et al. 1995] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. (1995). Specifying Distributed Software Architectures. In Schafer, W. and Botella, P., editors, *5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain. Springer-Verlag, Berlin.

[Manber 1999] Manber, U. (1999). *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, Reading, Massachusetts.

[Petri 1966] Petri, C. A. (1966). Kommunikation mit Automaten. *Technical Report RADC-TR-65-377, Griffiths Air Force Base, New York*, 1(1).

[Roch and Starke 1999] Roch, S. and Starke, P. (1999). Manual: Integrated Net Analyzer Version 2.2. *Humboldt-Universität zu Berlin, Institut für Informatik, Lehrstuhl für Automaten- und Systemtheorie*.

[Skillicorn and Talia 1998] Skillicorn, D. B. and Talia, D. (1998). Models and Languages for Parallel Computation. *ACM Computing Surveys*, 30:123–169.