Implementing Distributed Skeletons using D-Clean and D-Box *

Zoltán Horváth¹, Zoltán Hernyák², Viktória Zsók¹,

¹ Eötvös Loránd University, Faculty of Informatics
Department of Programming Languages and Compilers
H-1117 Budapest, Pázmány Péter sétány 1/C.
{hz, zsv}@inf.elte.hu

² Eszterházy Károly College
Department of Information Technology
H-3300 Eger, Eszterházy tér 1.
arom@ektf.hu

Abstract. The distributed evaluation of functional programs and the communication between computational nodes require high-level process description and coordination mechanism. D-Clean is a high-level functional language, which supports the distributed computation of Clean functions over a cluster. The lazy functional programming language Clean is extended by new language elements in order to achieve parallel features.

A process scheme defines a partial computation graph, where the nodes are functions to be evaluated and the edges are communication channels. The computational nodes are implemented as statically typed Clean programs. D-Clean is compiled to an intermediate level language called D-Box. The D-Clean generic constructs are instantiated into D-Box expressions. D-Box is designed for the description of the computational nodes.

The highest level the D-Clean coordination language primitives are applied for distributing the pure functional computation subtasks over a PC cluster. The distribution is made according to computation patterns, skeletons parameterized by the functions, types and data. This paper presents some of the classical skeletons written in the D-Clean and D-Box languages showing the appropriateness and applicability of the languages for designing distributed computation patterns.

1 Introduction

Parallel functional programming has already several years experiences background. Parallel programming in the Clean lazy functional programming language had several developing phases. At the very beginning there was a transputer version of the language [10]. The annotations [14] were used to specify...
which parts of a function should be evaluated in parallel. The parallel strategies [9] were based on annotations and we could define the evaluation order according to the structure of the results.

In a previously developed environment [2, 3] Clean programs could be interconnected via direct call of middleware services, making possible the asynchronous communication between the distributed functional components. It was recognised, that the distributed evaluation of functions and the communication between Clean programs need a higher level process description and control mechanism. For this reason a control language, the D-Clean language and an intermediate level language, the D-Box language, and their informal semantics were introduced in [1]. D-Clean contains high-level language elements for the coordination of the component functions in a distributed environment. Using these primitives applications can be written hiding the details of the application of the underlying middleware. In the appendix we summarize the introduced coordination structures (see section 7).

The definitions of higher order skeletons are given in D-Clean, which is a Clean-like language with distributed coordination language components. A skeleton is an abstract definition of the distributed computation and it is parameterized by functions, by type and by data. The computation patterns are identified and described by compositions of coordination elements. The coordination constructs have the role of manipulating and controlling the components written in Clean, which are expressing the pure computational aspects. These computational nodes have well identified subtasks of the original problem.

The coordination structures use channels for receiving the input data required for the arguments of their function expressions. The results (components of a k tuple in general case) of the function expression are sent to the output channels. Every channel is capable of carrying data elements of a specified base type from one computational node to another one. We use the unary algebraic type constructor $\text{Ch} \ a$ to construct channel types, where the base type $a$ is a transmissible type.

A coordination primitive usually has two parameters: a function expression (or a list of function expressions) and a sequence of input channels. The coordination primitives return a sequence of output channels. The signature of the coordination primitive, i.e. the types of the input and output channels are inferred according to the type of the embedded Clean expressions. In the appendix the $\text{aCh}$ denotes a channel type, while $\text{aCh}^*$ denotes a finite sequence of channel types.

A D-Clean program consists of a start expression, in which a collection of user-defined D-Clean process schemes can be applied. A process scheme itself is written in D-Clean too. The start expression is given as the $\text{DistrStart}$ function definition.

D-Clean coordination structures are mappings between communication channels and are designed as generic templates parameterized by types and by functions. The value of type parameters are determined by type inference. The templates are instantiated by the D-Clean pre-compiler at compile time.
The matching of types between the base types of channels and the types of embedded Clean expressions is a static semantic requirement. A D-Clean expression may be a compound expression or a direct use of coordination primitives. Process scheme definitions are named D-Clean expressions with formal parameters. A process scheme library can be built using the coordination primitives and the already defined schemes.

2 The D-Clean language

This section presents the introduced coordination primitives in an informal way. The figures of the section illustrate the working mechanism. $F$ denotes the function expression embedded into the coordination primitive.

**DStart** $\text{fun_expr} :: \text{aCh}^*$

The task of **DStart** primitive is to start the distributed computation by producing the input data for the dataflow graph. It has no input channels, only output channels. The results of the $\text{fun_expr}$ are sent to the output channels. Each D-Clean program contains at least one **DStart** primitive (see Figure 1).

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{DStart.png}
\caption{DStart node}
\end{figure}

**DStop** $\text{fun_expr} :: \text{aCh}^* \rightarrow < >$

When a function expression embedded into a **DStop** primitive has $k$ arguments, then the computation node evaluating the expression needs $k$ input channels. Each input channel carries one argument for the function expression.

The task of this primitive is to receive and save the result of the computation. It has as many input channels as the function expression requires, but it has no output channels. **DStop** closes the computational process. Each D-Clean program contains at least one **DStop** primitive (see Figure 2).

**DStop** is the last element of the D-Clean composition, the last element of the control flow. In some cases when the control flow contains forks, the network has multiple **DStop** elements.

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{DStop.png}
\caption{DStop node}
\end{figure}

**DApply** $\text{fun_expr} :: \text{aCh}^* \rightarrow \text{aCh}^*$
This variant of \texttt{DApply} applies the same function expression \( n \) times (see Figure 3/a) on \( n \times k \) channels. When the function expression has \( k \) arguments of types: \( t_1, t_2, ..., t_k \), the number of input channels is \( n \times k \). The types of the arguments periodically match the type of the channels:

\[
< Ch t_1, Ch t_2, ..., Ch t_k, Ch t_1, Ch t_2, ..., Ch t_k, Ch t_1, Ch t_2, ..., Ch t_k > .
\]

If the expression produces a tuple with \( m \) elements of the type \((p_1, p_2, ..., p_m)\), then the output channel sequence will contain \( m \times n \) elements, repeating the \( m \) type-sequences \( n \) times:

\[
< Ch p_1, ..., Ch p_m, Ch p_1, ..., Ch p_m, ..., Ch p_1, ..., Ch p_m > .
\]

\texttt{DApply} \(<\text{fun_expr}>\) :: \( \text{aCh}^* \rightarrow \text{aCh}^* \)

The second variant of \texttt{DApply} may apply different function expressions, which are given in the \(<\text{fun_expr}>\) sequence. The types and the number of the arguments of the function expressions can also be different. If the \(<\text{fun_expr}>\) sequence contains an identity function, then data received via the corresponding channel is directly forwarded to the next node.

The sequence of the input channels is constructed out of the channels required by the function expressions in the \(<\text{fun_expr}>\) sequence. The output sequence of channels is built up according to the results obtained by applying the function expressions. For example \texttt{DApply} \(<\text{F}_1, \text{id}, \text{F}_2, \text{F}_3>\) yields the structure presented in Figure 3/b.

\texttt{DFilter} \((a \rightarrow \text{Bool})\) :: \( \text{aCh}^* \rightarrow \text{aCh}^* \)

\texttt{DFilter} \(<a \rightarrow \text{Bool}>\) :: \( \text{aCh}^* \rightarrow \text{aCh}^* \)

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure3}
\caption{DApply variant a) and variant b).}
\end{figure}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure4}
\caption{A DFilter node}
\end{figure}
The DFilter primitive filters the elements of the input channels using a boolean function. It has two variants similarly to DApply. This is the D-Clean variant of the standard filter library function. This variant filters the incoming data elements before sending them to the outgoing channels.

DMap fun_expr :: aCh^* -> aCh^*

DMap <fun_expr> :: aCh^* -> aCh^*

DMap is a special case of DApply where the function expression must be an elementwise processable function \[9\]. It is the D-Clean variant of the standard map library function. It modifies the incoming data elements processing them one by one.

A valid parameter function expression for DMap can be a function expression either of type a->b or of type [a]->[b]. Suppose we have a list of n sublists as input data, then the \(qsort::![a]->[a]\) sorting function\[3\] is a valid function expression as parameter for DMap. It takes every sublist element of the input list and applies the parameter function expression, i.e. the \(qsort\) function on it. The result will be the list of the n sorted sub-lists.

**Fig. 5.** DMap nodes

DReduce fun_expr :: aCh^* -> aCh^*

DReduce <fun_expr> :: aCh^* -> aCh^*

DReduce is another special case of DApply with similar restrictions. A valid expression for DReduce has to decrease the dimension of the input channel type\[4\]. A valid expression has the type of form \([a]->b\). For example the \(sum::[a]->a\) function - which computes the sum of the elements of the input list - is a valid expression for DReduce.

DProduce fun_expr :: aCh^* -> aCh^*

DProduce <fun_expr> :: aCh^* -> aCh^*

DProduce is another special case of DApply. The expression has to increase the dimension of the channel type\[5\]. A valid expression must be of the form a->[b]. For example the \(divisors::Int->[Int]\) function - which generates all the divisors for an integer number - is a valid expression for a DProduce.

DDivideS fun_expr n :: aCh^* -> aCh^*

DDivideS is a static divider (see Figure 6). The expression splits the input data list into n parts and broadcasts them to n computational nodes. This

---

3 \(!\) denotes strict evaluation of the argument.
4 For example: list of lists \(\rightarrow\) list.
5 For example: list \(\rightarrow\) list of lists.
primitive is called static divider since the value of \( n \) must be known at pre-compile time.

The base type of the sublists has to be the same type as of the original list. Therefore the types of the output channels are the same as of the input ones. Consequently there will be \( n \) output channels.

![Fig. 6. DDivideS node](image)

\[
\text{DDriveS node}
\]

\[
\text{DMerge fun_expr :: aCh^* -> aCh^*}
\]

\text{DMerge} collects the input sublists from channels and builds up the output data lists. All the input channels must have the same type (see Figure 7).

![Fig. 7. DMerge node](image)

\[
\text{DMerge node}
\]

\[
\text{DLinear <fun_expr> :: aCh^* -> aCh^*}
\]

\text{DLinear} is a special coordination primitive. It simplifies the definition of the pipeline computation graph, where the nodes are connected to each other in a linear way (see Figure 8).

![Fig. 8. DLinear nodes](image)

\[
\text{DLinear nodes}
\]

\[
\text{DLinear <expr1, expr2, ..., exprk> is equivalent to the following composition of DMap primitives: (DMap expr_k) ... (DMap expr_2) (DMap expr_1).}
\]

3 The D-Box language

D-Clean is compiled to an intermediate level language called D-Box. The D-Clean generic constructs are instantiated into D-Box expressions. D-Box is designed for the description of the computational nodes which set a computational graph
D-Box expressions hide implementation details and enable direct control over the process-network. The asynchronous communication is based on language-independent middleware services.

The D-Box language is a description language for the source codes of computational nodes. In this language input and output protocols can be defined. A graphical developer environment was built to support a direct use of the D-Box language.

To describe this kind of graph, the nodes and the edges (connections) must be defined.

A computational node may use more than one input channel. At this level a channel identification mechanism is used. One input channel is described by its type and by the unique id of the channel. Notation $[T]$ is used in the type description of a channel, which is used to transfer a single list of elements of the base type $T$. Whenever a list of lists is sent via a channel, type $[[T]]$ is associated to it.

The input protocol also determines the synchronization mode of the input channels. There are three modes: memory, join1 and joink. The input is completely defined when the list of the input channels ($<INPUT_CHANNEL_LIST>$) and the input protocol ($INPUT_PROC_MODE$) are given.

The number and/or the base types of the input channels can be different from the types of arguments of the expression ($<ARGUMENT_TYPE_LIST>$). The matching of channel types to argument types is completed at code generation time according to the actual protocol. The same holds for the $<RESULT_TYPE_LIST>$ too.

The output protocol definition has the same structure as the input definition.
A complete D-Box definition has the following parts:

```
BOX <BOXID>
    { <SubGraphID>,
      { (<INPUT_CHANNEL_LIST>), INPUT_PROC_MODE },
      { (<ARGUMENT_TYPE_LIST>), <EXPRESSION>, (<RESULT_TYPE_LIST>) },
      { (<OUTPUT_CHANNEL_LIST>), OUTPUT_PROC_MODE } }
```

Fig. 10. A general structure of a D-Box with channels

It is possible to design the computational graph by using the D-Box Graphical Environment which is a Java application. This is a visual development kit. The expressions can be inserted into the boxes, and the protocols can be selected from a drop-down list. Adding ports to the boxes the channels can be defined easily.

The code generated by the compiler must be deployed and started on the computers of the cluster. This requires a kind of application environment. This environment provides the following services:

- (1) A local component of the middleware which provides the minimal operations so that the computer can join to the DBox cluster environment.
- (2) Operations for storing the executable files for a specified project. It is needed because by default there is no network file system on the windows platform. This facility acts as a kind of FTP server allowing to upload and download files, automatically identified by their project name, by the sub-GraphID, BoxID, channel type and by other parameters.
- (3) Name service to find the already started components.
(4) A scheduler to find an appropriate idle node to start a new component of the project.
(5) An optional component for monitoring the events of the environment.

Fig. 11. Using the D-Box Graphical Environment

Fig. 12. The structure of the Application Environment
4 Skeletons in D-Clean

In the examples presented in this section we will use an input generator for providing the input data in simple way. Usually in these examples for input we will consider list of integers, generated by the generator function. This can be easily extended to any data structure. The DStart primitive will take the input given by the generator and starts the distributed computation.

The other coordination primitive which must be included in any D-Clean skeleton is the DStop primitive. The terminate parameter function of the DStop primitive will gather the result and will save it. The WriteResult function is used for saving the final result into a file.

In the example we will use three functions as subtask of the computation nodes (f1, f2, f3), each of them has the type [Int] -> [Int].

The pipeline computation pattern is one of the common skeletons. It takes a list of functions and applies the composition of these functions on the input, see figure 13.

\[
\text{DStart} = (\text{DStop \ terminate}) (\text{DLinear function_list}) (\text{DStart generator})
\]

where
\[
\begin{align*}
\text{terminate} &= \text{WriteResult } "c:\DClean\resultlinear.txt" \\
\text{function_list} &= [f1, f2, f3]
\end{align*}
\]

The farm pattern divides the original task into subtasks, computes the subresults and builds up the final results from the subresults, see figure 14. The divide function divides the input into \( N \) parts, solve applies the qsort function on the divided inputs, combine will merge the subresults into a final one using the combine function. The complete code is given in the following:

\[
\text{DStart} = (\text{DStop \ terminate}) (\text{DMerge combine}) (\text{DApply solve}) (\text{DDivideS divider \ N}) (\text{DStart generator})
\]

where
\[
\begin{align*}
\text{divider} &= \text{divide } N \\
\text{solve} &= \text{qsort} \\
\text{combine} &= \text{combine_lists} \\
\text{terminate} &= \text{WriteResult } "c:\DClean\result.dat" \\
N &= 3
\end{align*}
\]

\[
\text{generator}::[\text{Int}]
\]
generator = [1, 9, 4, 6, 2, 8, 5, 3, 10, 7]

divide :: Int [Int] -> [[Int]]
divide n xs = [tak n (drop i xs) \ i<-[0..n-1]]
where
  tak n [] = []
  tak n [x:xs] = [x : tak n (drop (n-1) xs)]

combine_lists :: [[Int]] -> [Int]
combine_lists [] = []
combine_lists [x:xs] = merge x (combine_lists xs)

Fig. 14. The farm skeleton

An other version of the farm skeleton uses the DReduce primitive instead of DApply, see figure 15. By the sum function the dimension of the input will be reduced (i.e. the function has [a] -> b type).

DistrStart = (DStop terminate) (DReduce combine) (DApply solve)
             (DDivideS divider N) (DStart generator)
where
  divider = divide N
  solve = sum
  combine = sum
  terminate = WriteResult "c:\DClean\result.dat"
  N = 3

4.1 Composition of schemes

The following example takes the farm computation example, but as subtasks applies a pipeline scheme, see figure 16.

The square root values of the elements given by the generate function are computed using Newton iteration. The approximate square root of the value $a$ is calculated according to the following formula:
The generated real numbers are converted first to an ordered pair containing the value and the first iteration (the half of the value). The farm scheme is used for distributing the values among three pipelines. Each pipeline will iterate according to the above formula using the predefined pipeline scheme. After each pipeline finishes the iterations the combine function collects the subresults into a final list. At the end the result is saved into a file.

:: Pair = { d :: Real
          , a :: Real
        }

DistrStart = (DStop terminate) (DMerge combine) (DLinear function_list)
            (DDivideS divider N) (DStart generate)

where
generate = transform generator
divider = divide N
function_list = [ff, ff, ff, ff, ff, ff, ff, ff, ff, ff]
combine = combine_lists
terminate = WriteResult "c:\\DClean\\square_roots.dat"
N = 3
generator :: [Real]
generator = [1.0, 9.0, 4.0, 6.0, 2.0, 8.0, 5.0, 3.0, 10.0, 7.0]

\[ t :: \text{Real} \to \text{Pair} \]
\[ t x = \{d = x/2.0, a = x\} \]

\[ \text{transform} :: [\text{Real}] \to [\text{Pair}] \]
\[ \text{transform} x = \text{map} t x \]

\[ \text{divide} :: \text{Int} [\text{Pair}] \to [[[\text{Pair}]]] \]
\[ \text{divide} n xs = [\text{tak} n (\text{drop} i xs) \setminus i\leftarrow [0..n-1]] \]
\[ \text{where} \]
\[ \text{tak} n [] = [] \]
\[ \text{tak} n [x:xs] = [x : \text{tak} n (\text{drop} (n-1) xs)] \]

\[ \text{combine_lists} :: [[[\text{Pair}]]) \to [\text{Pair}] \]
\[ \text{combine_lists} x = \text{flatten} x \]

\[ f :: \text{Pair} \to \text{Pair} \]
\[ f x = \{d = 0.5*(((x.a/x.d)+x.d), a = x.a\} \]

\[ ff :: [\text{Pair}] \to [\text{Pair}] \]
\[ ff x = \text{map} f x \]

5 Related works

- PMLS and GpH are implicit parallel extensions of ML and Haskell respectively [16], on the other hand D-Clean uses explicit coordination structures. Opposed to skeleton based languages, D-Clean is designed to implement skeletons of distributed functional computations in the language itself.
- Eden [12, 11] extends Haskell to explicitly define parallel computation. Eden program consists of processes and uses communication channels, and the programmer has explicit control over communication topology. The execution is based on GHC implementation of concurrency, the run-time system controls sending and receiving messages, process placements and data distribution. On the other hand the middleware supporting the implementation of DClean and DBox languages is not language specific, components developed using other languages can be integrated into easily distributed applications.
- Nimo [18] is a visual functional dataflow language, supporting process networks. Nimo allows total graphic programming only, while DClean and DBox programs can be expressed in textual code form too. Nodes in Nimo are restricted for a fixed set of primitive operations of Haskell prelude, while in DClean nodes Clean expressions are allowed to achieve full power of functional programming at node level. Nimo does not support distributed computing, only concurrent execution is supported.
JoCaml is an extension of Objective Caml with primitives for network-transparent distributed and mobile programming [20] based on the join-calculus model instead of a pure data flow approach. Advanced discussion and survey of the dataflow languages can be found in [17]. Data oriented skeletons (like the farm skeleton) can be implemented using primitives which are quite similar to the primitives of dataflow languages.

6 Conclusion

The above examples provide a set of patterns, which demonstrates the usage of the introduced D-Clean coordination primitives. In our context a pattern serves as an easy to use computation skeleton in the distributed environment. It describes in a general manner a common algorithmic scheme parameterized by functions and data. The computation will be distributed over a network of computation nodes according to the description provided by the skeleton. The computation patterns are identified and described by compositions of coordination elements. The coordination constructs have the role of manipulating and controlling the components written in the functional language Clean, which are expressing the pure computational aspects. The presented set of skeletons demonstrates the applicability and compositibility of the D-Clean coordination language primitives for designing complex distributed computation patterns.

References


15. Dezső, Balázs: DBox Developer Environment, Project work documentation, Department of Programming Languages and Compilers, University Eötvös L., Budapest, Hungary, 2005.


7 Appendix - The D-Clean language reference

- **DStart fun expr :: aCh^*** starts the distributed computation.
- **DStop fun expr :: aCh^* <>** receives and saves the result of the computation.
- **DApply fun expr :: aCh^* -> aCh^*** applies the same function expression \(n\) times on \(n \cdot k\) channels.
- **DApply [fun expr] :: aCh^* -> aCh^*** applies different function expressions on the input channel list.
- **DFilter fun expr :: aCh^* -> aCh^*** filters the elements of the input channels using a boolean function.
- **DFilter [fun expr] :: aCh^* -> aCh^*** filters the elements of the input channels using boolean functions.
- **DMap fun expr :: aCh^* -> aCh^*** applies an elementwise processable function of type \(a \rightarrow b\) on channels.
- **DMap [fun expr] :: aCh^* -> aCh^*** applies a list of elementwise processable function of type \([a] \rightarrow [b]\).
- **DReduce fun expr :: aCh^* -> aCh^*** applies a function of type \([a] \rightarrow b\).
- **DReduce [fun expr] :: aCh^* -> aCh^*** applies a list of functions of type \([a] \rightarrow b\).
- **DProduce fun expr :: aCh^* -> aCh^*** applies a function of type \(a \rightarrow [b]\).
- **DProduce [fun expr] :: aCh^* -> aCh^*** applies a list of functions of type \(a \rightarrow [b]\).
- **DDivideS fun expr n :: aCh^* -> aCh^*** splits the input data list into \(n\) parts and broadcasts them to \(n\) computational nodes.
- **DMerge fun expr :: aCh^* -> aCh^*** collects the input sublists from channels and builds up the output input data list.
- **DLinear [fun expr] :: aCh^* -> aCh^*** simplifies the definition of the pipeline computation graph where the nodes are connected in a linear way.
- **DDivideN fun expr :: aCh^* -> :: aCh^*** is the dynamic version of the **DDivideS**, where the number of the threads \(N\) is calculated at run-time.

This article was processed using the \texttt{\LaTeX} macro package with LLNCS style