Communication Algebras

Michał Walicki [*] *Magne Haveraaen* *Sigurd Meldal*

Universitetet i Bergen
Institutt for Informatikk
HiB, 5020 Bergen, Norway
{michal, magne, sigurd}@ii.uib.no

Abstract

We introduce a semantic framework which generalises algebraic specifications by equipping algebras with descriptions of *evaluation strategies*. The semantic objects are not merely algebras, but algebras with associated data dependencies. The latter provide separate means for modeling computational concerns along with the functional specifications usually captured by an algebra. The formalization of evaluation strategies allows the formal manipulation of how the operations of an algebra are to be evaluated. This formalization introduces (1) increased portability among different hardware platforms, as well as (2) allowing a potential increase in execution efficiency, since a chosen evaluation strategy may be tailored to a particular platform. We present the development process where algebraic specifications are equipped with data dependencies, the latter are then refined and, finally, mapped to actual hardware architectures.

1 Introduction

The creed of algebraic specification is *abstraction*: one models programs as algebras, thus abstracting from the operational details and focusing on the abstract functionality of the system. A single such specification may be implemented on different platforms. The implementation design, as well as the particular platform on which it is realised, need not be the primary issues. It is one of the central tenets of the algebraic specification that the designer may initially model his system at a high level of abstraction without involving himself into low level implementation details, only introducing implementation concerns as appropriate, much later in the development process. At some point during the development process the choice of the set of constructors, the computation strategy, the target language, and, perhaps, the hardware platform begin to play a more important role. In particular, because of its intentional abstraction from the operational aspects, the standard algebraic approach to specification has difficulties when faced with the possibilities of parallel implementations. It is at this point that our intend method comes into successful play.

We propose a framework of *communication algebras* adressing the issue of how to incorporate computation-related aspects within the framework of classical algebraic semantics. Though occasionally tempting, we would like to avoid rebuilding the theory from scratch – we want to take advantage of the existing theoretical and tool-oriented results of algebraic specifications. Our aim is only to *extend* the classical framework with constructs which in an algebraic way addresses the choice of computational strategies.

This is achieved by associating additional structures, *data dependency* graphs, with standard algebras. The data depency graphs carry information about evaluation strategies for the operations of the algebras. We have to warn the reader that our data dependencies are not exactly what is meant by that in the literature, especially on imperative programs, e.g. [MW83, FOW87, KK90]. We proceed from the algebraic, that is, functional descriptions and dependencies relate functional

[*]The author gratefully acknowledges the financial support received from the Norwegian Research Council (NFR).

terms and not program statements. In particular, a variable is only a (sub)term denoting specific value – it is not an actual program variable of a program capable of storing various values at different points of computation.

We address mainly the semantic issues of this approach, without entering into a discussion of possible specification *formalisms*. Thus we will assume that data dependencies are given along with algebras, and study their interaction. Since data dependencies are standard algebraic objects we expect that they can easily be described in a standard algebraic fashion. (We intentionally refrain from including an analysis of control dependencies – unlike data dependencies, they seem to be inherent features of actual programs rather than of abstract data types.) In some cases data dependencies may be recovered directly from an algebraic specification and section 3 describes a way to do it.

By allowing computational structures into the semantics of a specification we provide the means not only for development of programs, but for the development of *efficient* programs. We will show that data dependencies provide such a means, enabling one to assess the time/space resources required and to pursue optimization strategies such as syntactic and semantic memoisation. Above all, the computational structures carry the information necessary to achieve *parallel* implementations; the identification of dependencies between various parts of a computation enables us to determine its possible parallel distributions. In addition to algebras and data dependencies, we will introduce a representation of (parallel) machine architectures and show how mappings from dependencies to architectures may be utilized to yield parallel implementation. The framework enables us then to *port the implementations of communication algebras* along the morphisms between various hardware architectures. It should be emphasized, however, that we are *not* concerned with a general theory of concurrency. In this context, we view concurrency merely as a means of increasing efficiency of the implementation.

Section 2 introduces the basic notions of communication algebras and their implementations and illustrates them on a simple example. Section 3 addresses the issue of constructing dependencies for a given algebra A given a specification satisfied by A. It illustrates the ideas of a programming language SAPPHIRE developed at the University of Bergen for which communication algebras provide the semantic framework. Section 4 discusses a series of specific issues which have not been treated in sufficient detail by the general framework of section 2, indicating some of its weaknesses. Section 5 summarises the results and indicates some directions for future work.

The paper focuses on the description of the concepts; its technical content is limited to a few results. It gives an improved version of the results from [WHM96]. The more technical aspects assume some elementary knowledge of the category theory (e.g., [BW90], [RB88]) with particular emphasis on fibrations.

2 The Framework of Communication Algebras

This section introduces gradually the framework of communication algebras. We define first some basic notions in 2.1, then data dependencies in 2.2, communication algebras in 2.3, and their implementation in 2.4. Subsection 2.5 summarizes these pieces presenting the view of the development process using communication algebras.

2.1 Signatures, Algebras and Graphs

We use the standard notion of a signature Σ as a pair $\langle \mathcal{S}, \mathcal{F} \rangle$ of sort and operation symbols. In subsection 4.2 we will use the category Sig with signatures as objects and *injective* signature morphisms.

For a signature Σ, $\mathsf{Alg}(\Sigma)$ will denote a category of Σ-algebras. It is "*a* category" because we do not make any assumptions as to what the algebras are, except that they are objects where one can interpret the Σ-terms. One may think of standard Σ-algebras with Σ-homomorphisms defined in the usual way. But one may as well think of the category of multilagebras with various multihomomorphisms, power algebras, unified algebras, etc. We will define various notions relatively to an arbitrary full subcategory $\mathsf{A} \subseteq \mathsf{Alg}(\Sigma)$.[1]

Definition 2.1 Gr is the category of *irreflexive directed acyclic graphs* where[2]:

1. objects are *DAGs* – algebras over the signature $\Gamma = \langle I; \_ \rightarrowtail \_ : I \times I \to Bool \rangle$ satisfying: $x \rightarrowtail^{\oplus} y \Rightarrow y \not\rightarrowtail^{\oplus} x$ (or equivalently $x \not\rightarrowtail^{\oplus} x$) ;
2. a Gr-morphism $g : G \to F$ is a mapping $I^G \to I^F$ such that $i \overset{G}{\rightarrowtail} j \Rightarrow g(i) \overset{F}{\rightarrowtail} g(j)$.

$i \rightarrowtail j$ indicates an edge between nodes i and j. For a relation R, $R^{\oplus}$ denotes its transitive (but not reflexive) closure.

For aesthetic reasons, we will allow the graph with the empty carrier, $\emptyset$, which is the initial object in Gr. We will consider "graph" a synonym for "DAG" since DAGs are the only graphs we will talk about. Also, we will usually not distinguish between a graph A and its relation $\overset{A}{\rightarrowtail}$. $A^{\oplus}$ denotes the transitive closure, *i.e.,* the graph $\langle I^A, (\overset{A}{\rightarrowtail})^{\oplus} \rangle$. In particular, a graph A will be said to be *well-founded* when the relation $\overset{A}{\rightarrowtail}$ is.

We may occasionally use the following terms:

- A graph is *sequential* if $\rightarrowtail^{\oplus}$ is a total ordering of the nodes.
- A *partitioning* of a graph is a partitioning of the set of nodes $I = \bigcup_{k=1}^{k=n} I_k$ with $1 < n < \omega$ such that each I_k is sequential.
- An *st-indexed* ("space-time") graph has nodes indexed by $\langle s, t \rangle \in S \times \mathbb{N}$, such that:
  1. $\forall s, t : \langle s, t \rangle \rightarrowtail \langle s, t+1 \rangle$;
  2. $\forall s, s', t, t' : \langle s, t \rangle \rightarrowtail \langle s', t' \rangle \Rightarrow t' = t + 1$;
  3. $\exists k \in \mathbb{N} : \forall s, s', t : \langle s, t \rangle \rightarrowtail \langle s', t+1 \rangle \Leftrightarrow \langle s, t+k \rangle \rightarrowtail \langle s', t+k+1 \rangle$.

The *st*-indexed graphs are singled out because they will be used as the **s**pace-**t**ime representation of hardware architectures [MW83]. The s component identifies a processor and the t component a time-point during a computation. Edges represent possible communications. The first condition says that each processor performs a sequential computation with the possibility of maintaining its local data from one step of the computation to the next one. The second condition requires all elementary communications to happen in a single time-step. The last condition says that the communication topology is fixed throughout the whole computation, i.e., the graph is actually a repetition of the pattern of the k-steps communications.

We will make extensive use of a different notion of a morphism between graphs F and G which amounts to *embedding* of F into the *path closure* of G, mapping F-nodes to G-nodes and F-edges to G-paths.

A path $i \rightarrowtail^{+} j$ in a graph is a sequence of nodes $k_1, ..., k_z$ with $z > 1, i = k_1, j = k_z$ such that for $1 \leq n < z : k_n \rightarrowtail k_{n+1}$. $\rightarrowtail^{+}$ denotes all the paths generated by $\rightarrowtail$ and, for a given graph G, its path closure is a multigraph $G^{+} = \langle I^G, (\overset{G}{\rightarrowtail})^{+} \rangle$. Obviously, in general, $G^{\oplus} \neq G^{+}$.

Definition 2.2 Gr^{+} is the category where:

1. objects $Obj(\mathsf{Gr}^{+}) = Obj(\mathsf{Gr})$;
2. a morphism $m : F \to G$ is a mapping $m : I^F \to I^G$ such that $i \overset{F}{\rightarrowtail} j \Rightarrow m(i)(\overset{G}{\rightarrowtail})^{+} m(j)$;
3. the composition of morphisms is defined pointwise in the obvious way:

   for $f : F \to G, g : G \to H$, $f \circ g(i) = g(f(i))$ and if $f(i \rightarrowtail j) = f(i)(\overset{G}{\rightarrowtail})^{+} f(j)$, then $f \circ g(i \rightarrowtail j)$ is obtained by the iterated application of g to all the edges in the path $f(i)(\overset{G}{\rightarrowtail})^{+} f(j)$.[3]

Example 2.3 An example of a morphism $f : D \to W$ in Gr^{+} is given in the figure 1.

[1] Using the different font for categories, we hope that the use of the subset symbol $\subseteq$ also for subcategory won't create any confusion.

[2] We assume fixed sort *Bool* with two elements.

[3] Composition is written in the diagrammatic order: $a \circ b$ means "first a, then b".

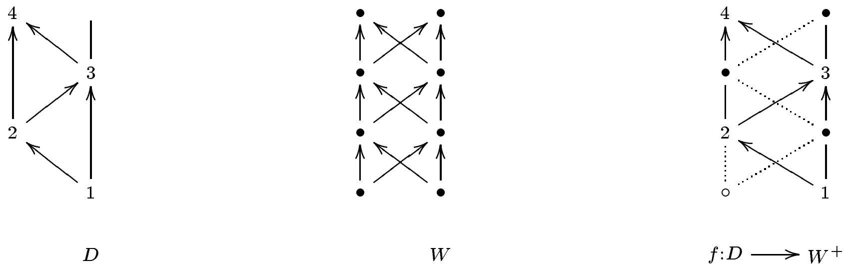


Figure 1: A morphism in Gr^+.

□

We note the following characterisation:

Proposition 2.4 *Let $e : A \to B \in Mor(\mathsf{Gr}^+)$. e is mono $\Leftrightarrow$ e is injective on the nodes.*

Proof.

$\Rightarrow$) If e is mono, then for any T and two arrows $T \overset{f}{\underset{g}{\rightrightarrows}} A$, $f \circ e = g \circ e \Rightarrow f = g$. In particular, for any node $i \in I^T : e(f(i)) = e(g(i)) \Rightarrow f(i) = g(i)$. Taking T as a one-node graph and considering all $f, g : T \to A$, implies that e must be injective on the nodes.

$\Leftarrow$) Suppose e is injective on the nodes and $f \circ e = g \circ e$. For each node $i \in I^T$ we then have $e(f(i)) = e(g(i)) \Rightarrow f(i) = g(i)$. To verify that $f = g$, we have to check that also for each edge k in T, $f(k) = g(k)$. If not, then there exists a node $j \in I^A$ which is on the path $f(k)$ but not $g(k)$ (or other way around). Since e is injective on the nodes, $e(j)$ cannot be on the path $e(g(k))$, that is, $e(g(k)) \neq e(f(k))$,[4] contrary to the assumption $f \circ e = g \circ e$.

□

2.2 Data Dependencies

As remarked in introduction, our data dependencies relate functional terms and not program statements. Thus, for instance, term $f(s(x), 2)$ may depend on $s(x)$ but x is just a variable which, in any instance, has some specific value. The variable x never depends on itself, as it might do if it was a program variable occurring in a statement $x := x + 1$.

For a signature Σ, a Σ *data dependency* is a graph with nodes labeled by ground Σ-terms $\mathcal{T}_\Sigma$. The only requirement we put on the morphisms is that they respect the sorting of labels.

Definition 2.5 $\mathsf{DD}(\Sigma)$ is the category of Σ *data dependencies* where (Γ is as in definition 2.1):

1. an object is a Σ data dependency – an algebra A over the signature $\Gamma_\Sigma = \Gamma \cup \{lab : I \to \mathcal{T}_\Sigma\}$ such that the reduct $A|_\Gamma$ is an object in Gr;
2. a $\mathsf{DD}(\Sigma)$-morphism $m : C \to D$ is a Gr^+-morphism between the underlying graphs $Gr(C) \to Gr(D)$, such that $\forall i \in I^C : Sort(lab^D(m(i))) = Sort(lab^C(i))$.[5]

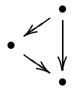
In the sequel, we will let D denote an arbitrary subcategory of $\mathsf{DD}(\Sigma)$. $Gr : \mathsf{DD}(\Sigma) \to \mathsf{Gr}^+$ is the obvious forgetful functor which, for a given Σ data dependency D, returns its underlying graph. We call such a graph the *shape of D*.

[4]Injectivity on the nodes implies injectivity on the edges (and paths), but the opposite implication does not hold.

[5]For a signature $\Sigma = \langle \mathcal{S}, \mathcal{F} \rangle$, the function $Sort : \mathcal{T}_{\Sigma,X} \to \mathcal{S}$ returns the sort of the argument term.

The relation $i \rightarrowtail j$ is ment to indicate that the computation at node j depends on the computation at node i. If $lab(i) = s$ and $lab(j) = t$ this means that the result of the evaluation of t at j depends on the result of the evaluation of s at i.

The dependency morphism $C \to D$ expresses the fact that D can *simulate* C – dependencies of C are reflected in D, not necessarily as direct edges but possibly as paths, and they may be enriched with new dependencies.

Remark. Gr-embedding of A into B is a special case of a $\mathsf{DD}(\Sigma)$-morphism $A \to B$. One might expect that a surjective (on the nodes) Gr-morphism $B \to A$ would yield a simpler notion of simulation. However, it would require our graphs to be reflexive (this is not a big problem) and, more importantly, would exclude many cases where B is a richer structure than A. For instance, there is no surjective Gr-morphism from $B =$  onto $A = \bullet \rightarrowtail \bullet \rightarrowtail \bullet$. But we would like to say that B does reflect the dependencies of A (and introduces some more) – and so there is a $\mathsf{DD}(\Sigma)$-morphism $A \to B$. □

Example 2.6 To give a flavor of our aims, consider a specification of the Fibonacci function:

$$\begin{array}{lll}
\textsc{Fib}\ \textbf{is}\ \mathcal{S}: & \mathbb{N} & \\
\qquad\ \mathcal{F}: & \ldots\text{the usual definitions of } \mathbb{N} & \\
 & F: \mathbb{N} \to \mathbb{N} & \\
\qquad\ \mathcal{A}: & F(0) = 1 & \\
 & F(1) = 1 & \\
 & F(n+2) = F(n) + F(n+1) &
\end{array}$$

Ignoring for the moment details (to be discussed later), we may easily recognize the dependency given by the tree of recursive calls R in figure 2. Certainly, definition 2.5 allows us to map this tree, for instance, onto a linear order (respecting the ordering in the tree and avoiding loops). In the following subsection we will couple dependencies with algebras and thus specify the means of avoiding uniteresting morphisms admitted, in principle, by definition 2.5. In the figure 2 we have indicated two dependecy morphisms: d' corresponding to syntactic memoisation (identifying the nodes with identical labels), and d'' corresponding to semantic memoisation (identifying $F(0)$ with $F(1)$) given the above specification.[6] Here, we may consider the dependency D as a refinement

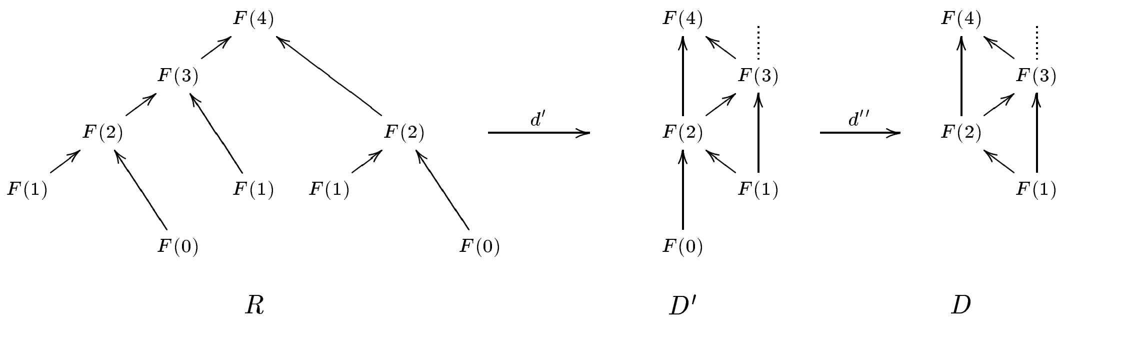


Figure 2: The recursive tree dependency R, together with the "actual" dependencies for computation of Fibonnaci, D' and D, and the dependency morphisms $d = d'' \circ d'$.

(improvement) of R. Notice that D is sequential under the ordering $F(n) < F(m)$ for $n < m$. Nevertheless, the partitioning of D indicated by the vertical arrows implies serious savings of storage space, even if the computations along both lines are heavily synchronised and far from independent. □

[6] In subsection 4.3.1, we suggest a more specific notion of a dependency morphism *compatible* with a given class of algebras of which d'' is an example.

Remark. As a matter of fact, there is also a dependency morphims $g : D \to R$ which embeds D into the leftmost part of R^+. This illustrates the fact that refinement of data dependencies is not necessarily an "improvement" in the narrow sense of memoisation – it rather reflects the relation of one dependency being able to perform the computations of which another one is capable. □

2.3 Communication Algebras

A Σ-communication algebra is a Σ algebra with an associated Σ-dependency. The latter provides a clue for how various operations of the algebra are to be computed. In general, for $\mathsf{A} \subseteq \mathsf{Alg}(\Sigma)$, $\mathsf{D} \subseteq \mathsf{DD}(\Sigma)$, an (A, D)-communication algebra is an A-algebra with an associated D-dependency.

Definition 2.7 An (A, D) *communication algebra* C is a triple $\langle \bar{C}, \vec{C}, ev^C \rangle$ where

1. $\bar{C} \in Obj(\mathsf{A})$;
2. $\vec{C} \in Obj(\mathsf{D})$;
3. $ev^C : I^{\vec{C}} \to \bar{C}$ is an evaluation function such that $lab^{\vec{C}}(i) = t \Rightarrow ev^C(i) = t^{\bar{C}}$.

The ev function is uniquely determined by $\bar{C}$ and the labelling of $\vec{C}$, so we will usually write a communication algebra as a pair $\langle \bar{C}, \vec{C} \rangle$. When (A, D) does not matter or is clear from the context, we will say C-algebra (instead of (A, D) algebra).

Dependencies may (and will) be interpreted as the requirements on the communications between different space-time points on an actual architecture. Consequently, we will refer to the dependency part $\vec{C}$ of an algebra C as its *communication* part.

Definition 2.8 An (A, D) *homomorphism* $h : C \to D$ is a pair $\langle \bar{h}, \vec{h} \rangle$ such that:

1. $\bar{h} : \bar{C} \to \bar{D} \ \in Mor(\mathsf{A})$;
2. $\vec{h} : \vec{D} \to \vec{C} \ \in Mor(\mathsf{D})$, and
3. $\bar{h}(ev^c(\vec{h}(i))) = ev^D(i)$, for all $i \in I^{\vec{D}}$.[7]

$$\begin{tikzcd} \vec{C} \arrow[r, dotted, "ev^C"] \arrow[dr, phantom, "\circlearrowleft"] & \bar{C} \arrow[d, "\bar{h}"] & C \arrow[d, "h"] \\ \vec{D} \arrow[u, "\vec{h}"] \arrow[r, dotted, "ev^D"] & \bar{D} & D \end{tikzcd}$$

Figure 3: A homomorphism between two communication algebras.

Like for C-algebras, when (A, D) does not matter, or is clear from the context, we will speak about C-homomorphisms.

$\mathsf{CAlg}(\mathsf{A}, \mathsf{D})$ is the category with (A, D) algebras as objects and (A, D) homomorpisms as morphisms (and obvious pointwise composition). When talking about the entire category $\mathsf{CAlg}(\mathsf{Alg}(\Sigma), \mathsf{DD}(\Sigma))$, we will instead write $\mathsf{CAlg}(\Sigma)$. The functor $Dep : \mathsf{CAlg}(\mathsf{A}, \mathsf{D}) \to \mathsf{D}^{op}$ projects C-algebras onto their communication part, i.e., $Dep(A) = \vec{A}$ and $Dep(h) = \vec{h}$.[8]

Revisiting the example 2.6 and dependency morphisms from figure 2, it is obvious that given, for instance, the standard algebra of natural numbers N with function F, we may form C-algebras $\langle N, R \rangle$, $\langle N, D' \rangle$ and $\langle N, D \rangle$ with the respective morphisms, e.g. $\langle id_N, d \rangle : \langle N, D \rangle \to \langle N, R \rangle$. On the other hand, a dependency morphism which sends, for instance, a node labeled with $F(2)$ to a node labeled with $F(4)$ will not yield a C-morphism unless some additional (and here, surprising) identifications take place in the algebra part N.

[7]The diagram 3 commutes, but not in a category (at least none we have defined explicitly), and so we write the composition of functions explicitly as $a(b(i))$.

[8]Of course, each (A, D) has its own functor $Dep_{(\mathsf{A},\mathsf{D})}$. But we may as well view each such a functor as a restriction of $Dep_{\mathsf{CAlg}(\Sigma)}$ to the respective source subcategory.

2.4 Machines and Implementations

We introduce the model of machine architectures and their mappings (simulations) in 2.4.1, and then define implementation of a communication algebra on a given architecture 2.4.2.

2.4.1 Machine architectures

In order to relate the dependencies to hardware architectures we will consider the latter as space-time unfoldings of the actual inter-processor communication structure [MW83]. Thus, a (static) architecture is simply an *st*-indexed DAG, where each node represents a unique time-point at a given processor and edges represent the possible single-step communications. This perspective may be extended to cover dynamically changing architectures as well (it would merely require more general graphs than the *st*-indexed ones) but for the moment we restrict our attention to the static case.

A morphism between two architectures expresses the possibility of using one architecture, Z, to *simulate* another, W, i.e., that each communication of which W is capable can be simulated by a chain of communications in Z. This is captured by the existence of an embedding of W into the path closure of Z, that is, a Gr^+-monomorphism $W \to Z$. Consequently, we define:

Definition 2.9 The category of *machine architectures* is $\mathsf{MAr} = \mathsf{Gr}^+$.

Thus, similarly to the dependency morphisms, an architecture morphisms $W \to Z$ tells us that, and how, Z can simulate communications of W.

Example 2.10 An n-dimensional vector W has n processors connected so that each processor, except for W_1 and W_n, has two-ways communication channels to its left and right neighbour: $W_1 \rightleftarrows W_2 \rightleftarrows \cdots \rightleftarrows W_n$. A space-time unfolding for a 2-dimensional vector is illustrated in figure 4.a). The vertical arrows $(W_i, k) \rightarrowtail (W_i, k+1)$ are added to represent the fact that each processor can carry its local data from one computation step to the next.

A ring Z with 3 processors where, in each step, processor Z_1 can communicate to Z_2, Z_2 to Z_3, and Z_3 to Z_1 is given in figure 4.b). This architecture may simulate the 2-dimensional vector W by the morphism m indicated in figure 4.c). □

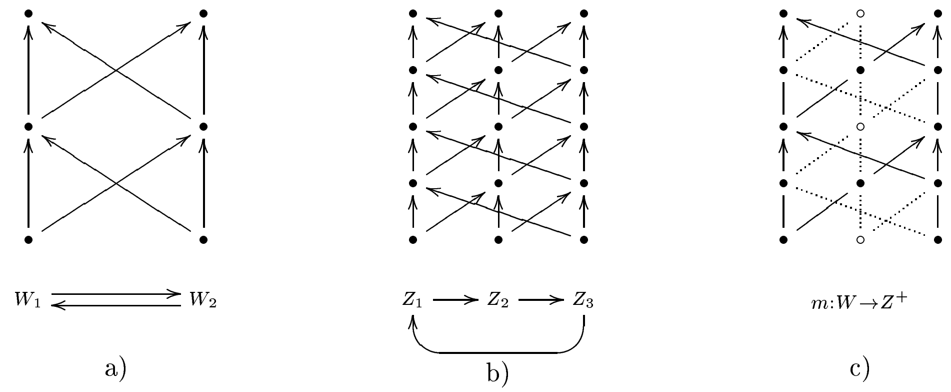


Figure 4: Space-time representations of a) 2-dimensional vector W, b) 3-processor ring Z, and c) a simulation m of W by Z.

We might have given a more restrictive definition of MAr. For instance, the unfoldings of actual processor-architectures will be *st*-indexed graphs which are simply repeating patterns of one-step communications. Also, simulation morphisms might be restricted to Gr^+-monomorphisms. Nevertheless, we choose this more generous definition because it will result in a more uniform treatment – it will allow us to consider also shapes of data dependencies as objects, and implementations as morphisms of the same category MAr.

2.4.2 Implementation

The composition of functors $Dep \circ Gr^{op} : \mathsf{CAlg}(\mathsf{A},\mathsf{D}) \to \mathsf{D}^{op} \to \mathsf{MAr}^{op}$ applied to an algebra returns the underlying graph of its communication part (which is an object of the category $\mathsf{MAr} = \mathsf{Gr}^{+}$). Implementation of an algebra $\langle \bar{C}, \vec{C} \rangle$ on a given architecture W amounts to a simulation of the (shape of the) communication structure given by $Gr(\vec{C})$ on W. That is

Definition 2.11 An *implementation* of a communication algebra C on an architecture W is an MAr-morphism $m : (Dep \circ Gr^{op})(C) \to W$.

An implementation, so to speak, maps the dependency relation $\rightarrowtail$ to the communication structure $\rightarrowtail$ of a hardware architecture.

Example 2.12 The dependency $\vec{D}$ (with the shape D from figure 2) is mapped onto two-processors vector W as it was illustrated in example 2.3, figure 1. Figure 5 is just a repetition of this earlier example: a) is the data dependency $\vec{D}$ with the shape $Gr(D)$, b) is a two-processors vector W and c) is an implementation f of D on W.

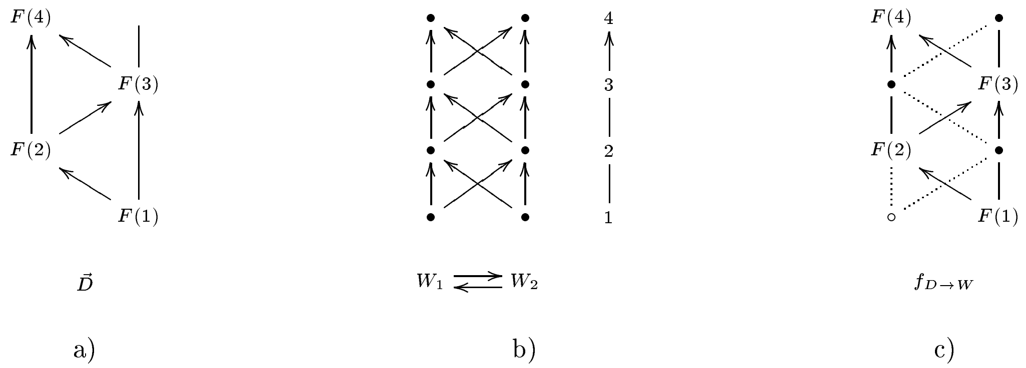


Figure 5: a) The data dependency $\vec{D}$, b) space-time representation of a 2-dimensional vector W, and c) an Gr^{+}-embedding $f : Gr(D) \to W^{+}$.

□

Combining such implementation morphisms with the simulation morphisms between various architectures, leads to the possibility of porting the implementations.

Example 2.13 The mapping f from figure 5.c) is an implementation of a communication algebra $D = \langle \bar{D}, \vec{D} \rangle$ (where $\bar{D}$ is a Σ_{FIB}-algebra satisfying FIB) on the vector W. Then, the mapping m (figure 4.c) gives an implementation of D on the ring Z. The obvious results are shown in figure 6.

□

Since an implementation m is a mapping of $Gr(\vec{A})$ into W^{+}, it will be sometimes written explicitly as $m_{A\to W}$ and called an *implementation morphism* or just an *implementation*. Unfortunately, an implementation morphism is not a morphism between implementations. The latter is defined as follows:

Definition 2.14 A morphism between two (A,D) implementations, $m : f_{A\to W} \to g_{B\to Z}$, is a pair $\langle\langle \bar{m}, \vec{m} \rangle, m^{+} \rangle$, where:

1. $\langle \bar{m}, \vec{m} \rangle : A \to B \in Mor(\mathsf{CAlg}(\mathsf{A},\mathsf{D}))$;
2. $m^{+} : Z \to W \in Mor(\mathsf{MAr})$, and
3. $Gr(\vec{m}) \circ f = g \circ m^{+}$.

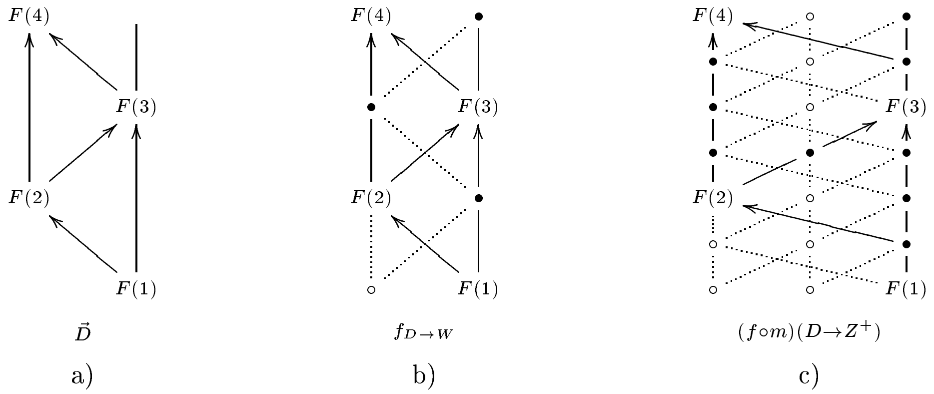


Figure 6: a) Data dependency, b) its implementation on a 2-dimensional vector W, and c) implementation b) ported to the ring Z

$$
\begin{tikzcd}
\bar{A} \arrow[d, "\bar{m}"'] & \vec{A} \arrow[l, dotted, "ev^A"'] \arrow[r, "f"] \arrow[dr, phantom, "\circlearrowleft"] & W & f_{A\to W} \arrow[d, "m"] \\
\bar{B} & \vec{B} \arrow[l, dotted, "ev^B"'] \arrow[u, "\vec{m}"] \arrow[r, "g"'] & Z \arrow[u, "m^+"'] & g_{B\to Z}
\end{tikzcd}
$$

Figure 7: A homomorphism between two implementations.

Simplifying a bit, we may depict the situation as in figure 7.
The diagram of figure 7 is a convenient abbreviation ignoring the application of Gr. The commutativity of its rightmost square is actually the requirement of the commutativity (in Gr^+) of the rightmost square in the diagram from figure 8.

$$
\begin{tikzcd}
\vec{A} \arrow[r, mapsto, "Gr"] & Gr(\vec{A}) \arrow[r, "f"] & W \\
\vec{B} \arrow[u, "\vec{m}"] \arrow[r, mapsto, "Gr"'] & Gr(\vec{B}) \arrow[u, "Gr(\vec{m})"] \arrow[r, "g"'] \arrow[ur, phantom, "\circlearrowleft"] & Z \arrow[u, "m^+"']
\end{tikzcd}
$$

Figure 8:

We let $\mathsf{Imp}(\mathsf{A},\mathsf{D})$ be the category with objects being implementations of (A,D) algebras from definition 2.11, with morphisms defined in definition 2.14 and pointwise composition.[9] The forgetful functor $MAr : \mathsf{Imp}(\mathsf{A},\mathsf{D}) \to \mathsf{MAr}^{op}$ is the obvious projection of implementations onto their architecture part.[10]

2.5 The Development Process

The framework is motivated by the view of the development of (possibly parallel) implementations by means of communication algebras. We envision the process starting with the usual refinement of

[9] This works because the commutativity of the square in figure 8 – i.e., for each edge e in $Gr(\vec{B})$, the paths obtained in W by $m^+(g(e))$ and by $f(Gr(\vec{m})(e))$ are identical – implies commutativity for each path in $Gr(\vec{B})$. Hence composition with another such commuting square, given by $Gr(\vec{n}) : Gr(\vec{C}) \to Gr(\vec{B})$, $n^+ : Y \to Z$, and $h : Gr(\vec{B}) \to Y$, will yield a commuting square $(Gr(\vec{n}) \circ Gr(\vec{m})) \circ f = h \circ (n^+ \circ m^+)$.

[10] Again, we may view each $MAr_{(\mathsf{A},\mathsf{D})}$ as a restriction of $MAr_{\mathsf{CAlg}(\Sigma)}$ to $\mathsf{CAlg}(\mathsf{A},\mathsf{D})$.

algebraic specifications (an algebra $\bar{A}$ in figure 9). At some point, when the specification becomes sufficiently concrete, one introduces the dependencies, i.e., constructs a communication algebra $\langle \bar{B}, \vec{B} \rangle$. Dependencies can then be refined along with a further refinement of the algebra itself or, perhaphs, keeping the algebra fixed.

Remark. Since there is no surjectivity requirement on ev, the communication part $\vec{C}$ need not contain the full information about all operations of the algebra $\bar{C}$. (In the extreme case, the shape of $\vec{C}$ might be the empty graph.) There are several reasons for the lack of such a requirement. Firstly, it allows us to consider a standard algebra $\bar{C}$ as a communication algebra with an empty communication part. More importantly, it allows us to model the development process in which more and more operations of the algebra are gradually equipped with more detailed computational information. Finally, we may imagine a specification of C-algebras as a kind of algebraic specification with hidden functions: only the visible operations are of interest to the implementation, and the resulting C-algebra need only provide the dependency structure for these operations. $\square$

When all the required operations of the algebra have been associated with the satisfactory dependencies, $\langle \bar{C}, \vec{C} \rangle$, one may choose an architecture on which to implement the obtained communication algebra, $f_{C \to W}$. Once this is done, there remains only the question of portability, that is, of moving the implementation to different architectures.

One may tend to imagine that a transition from one level (category) to the next means that the structure resulting from the development at the previous level is complete and remains fixed for the rest of the process. Thus, having reached the algebra $\bar{B}$ we begin to introduce and refine dependencies while $\bar{B}$ remains unchanged (and so, $\bar{C}$ is actually $\bar{B}$). Similarly, having designed $\langle \bar{C}, \vec{C} \rangle$ and implementing it on W, we may move to another architecture Z but $\langle \bar{C}, \vec{C} \rangle$ will be kept fixed (i.e., $\langle \bar{D}, \vec{D} \rangle = \langle \bar{C}, \vec{C} \rangle$). Although this is probably the most natural scenario, figure 9 illustrates the process in its full generality, where *all* the aspects may change at *all* levels.

$$
\begin{array}{lccccccc}
 & \underline{\mathsf{Imp}(\mathsf{A},\mathsf{D})} & & \underline{\mathsf{CAlg}(\mathsf{A},\mathsf{D})} & & \underline{\mathsf{A}} & & \\
\mathsf{A} & \bar{D} \rightarrow \cdots \rightarrow & & \bar{C} \rightarrow \cdots \rightarrow & & \bar{B} \rightarrow \cdots \rightarrow & & \bar{A} \\
 & \uparrow ev^D & & \uparrow ev^C & & \uparrow evB & & \\
\mathsf{D}^{op} & \vec{D} \rightarrow \cdots \rightarrow & & \vec{C} \rightarrow \cdots \rightarrow & & \vec{B} & & \\
 & \downarrow g & & \downarrow f & & & & \\
\mathsf{MAr}^{op} & Z \rightarrow \cdots \rightarrow & & W & & & &
\end{array}
$$

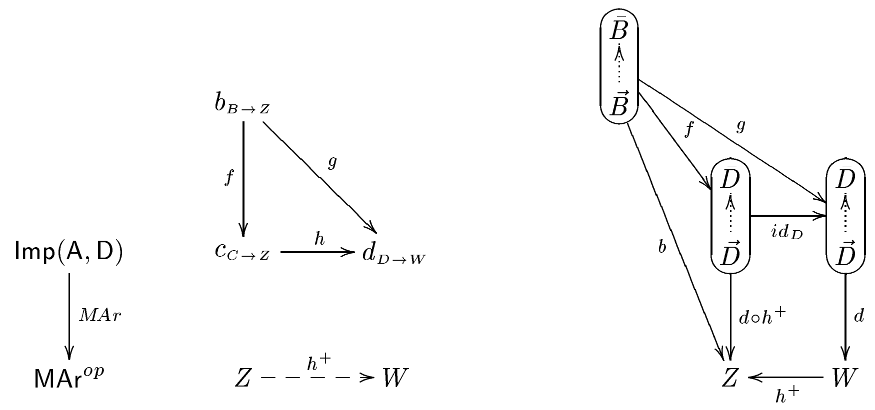
Figure 9: The development process

Although the technical results below (and in section 4) are rather straight-forward and hardly surprising, they provide a sound justification for the above methodology as well as a clue as to how it may be applied. They yield a pleasing conceptual structure in which algebras can be thought of as indexed by various dependencies, and communication algebras by machine architectures. More importantly, they give us the possibility of modularising the implementation process and reusing its results:

The (simulation) mappings between architectures MAr may be defined and stored independently of the actual programs implemented on them. Suppose that we have implemented a communication algebra C on an architecture W, $f_{C \to W}$. If MAr is a fibration with an explicitly defined cleavage cl, then all we need to do to port C to another architecture Z is to find (that is, choose from the library) an MAr morphism $W \to Z$. The cleavage then yields an algebra D together with an implementation $g_{D \to Z}$ which is an implementation of $f_{C \to W}$. As a matter of fact, the triviality of the proof (that MAr is a fibration) coincides with the desired fact that $D = C$.

Lemma 2.15 *For any* (A, D), $MAr : \mathsf{Imp}(\mathsf{A}, \mathsf{D}) \to \mathsf{MAr}^{op}$ *is a split fibration.*

Proof. We use the notation from the diagram to the right which illustrates the fibration situation in the present context (dashed arrows indicate the directions in the $\_^{op}$ categories):



For an MAr^{op}-morphism $h^+ : Z \dashrightarrow W$ and implementation $d = d_{D\to W}$, the cleavage is defined as

$$mcl(h^+, d) \stackrel{\text{def}}{=} h : c \to d \ where \begin{cases} h = \langle id_D, h^+\rangle \\ c = d \circ h^+ : Gr(\vec{D}) \to Z^+ \end{cases} \tag{1}$$

It is trivially a morphism in $Mor(\mathsf{Imp(A,D)})$ since $id_{\vec{D}} \circ c = d \circ h^+$.

Let $g : b \to d$ be $\langle\langle \bar{g}, \vec{g}\rangle, h^+\rangle$ (i.e., such that $MAr(g) = h^+$). The unique $f : b \to c$ with $MAr(f) = id_Z$ making $f \circ h = g$ is then $\langle\langle \bar{g}, \vec{g}\rangle, id_Z\rangle$. $f \in Mor(\mathsf{Imp(A,D)})$ because $\vec{g} \circ b = d \circ h^+ = c \circ id_Z$, where the first equality holds since $g \in Mor(\mathsf{Imp(A,D)})$.

Composition of cleavages yields a cleavage: for $h^+ : W \to Z$, $g^+ : Z \to V$ and $d \in MAr^{-1}(W)$ we have $mcl(h^+, d) = \langle id_D, h^+\rangle$, $mcl(g^+, d \circ h^+) = \langle id_D, g^+\rangle$, so $mcl(h^+, d) \circ mcl(g^+, d \circ h^+) = \langle id_D, h^+ \circ g^+\rangle = mcl(h^+ \circ g^+, d)$. Hence MAr is a fibration.

It is split: we have $(id_W)^*(d) = d$, so the natural transformation $id_{MAr^{-1}(W)} \rightrightarrows (id_W)^*$ is identity. Since composition of cleavages gives a cleavage, we have that $(g^+)^* \circ (h^+)^* = (h^+ \circ g^+)^*$. $\square$

The fiber over a given architecture represents naturally all communication algebras which can be implemented on this architecture, together with the associated implementation morphisms.

Example 2.16 The collection of all our examples so far illustrates exactly this idea of development. We have designed a dependency $\vec{R}$ which was then refined to $\vec{D}$. The latter was implemented on W (figure 5) and ported to Z (figure 6). Provided that the mappings m and f were available from a library, all the work we had to do was to design the mapping d. $\square$

$$\begin{tikzcd}
\vec{D} \arrow[d, dotted, "f \circ m"'] & \vec{D} \arrow[l, dotted, "id_D"'] \arrow[d, dotted, "f"] & \vec{R} \arrow[l, "d"'] \\
Z & W \arrow[l, dotted, "m"] &
\end{tikzcd}$$

Figure 10: The summary of the example

3 Constructing Graphs and Labelings

So far we have considered relationships between various aspects – functional specification (algebra), evaluation (communication part) and implementation – given independently from each other. In this section we show how, given an algebra satisfying some specification, one can *derive* an efficient data dependency and a labelling function. First, in 3.1, we introduce a general way of defining data dependencies for a given signature and a set of nodes, by defining a context free language for every node of the graph. This grammar induces the edges between and the labelings of the given set of nodes. The next subsection 3.2 illustrates the use of this general idea with particular specifications of recursive functions. In terms of figure 9, we are at the point $\langle \bar{B}, \vec{B} \rangle$, have a specific algebra $\bar{B}$ and construct the communication part $\vec{B}$ for it.

3.1 From signature to data dependency

Recall that a data dependency D, Definition 2.5, for a given signature Σ consists of a set of nodes I^D, a set of directed arcs given by $\rightarrowtail^D$, and a labeling function lab^D from nodes to terms $\mathcal{T}_\Sigma$.

Definition 3.1 A *context free grammar* G is represented by a 4-tuple $\langle T, N, \longrightarrow, s \rangle$, where T is the set of terminal symbols, N is the set of non-terminal symbols, $N \cap T = \emptyset$, $\longrightarrow \subseteq N \times (N \cup T)^*$ is the set of production rules, and $s \in N$ is the start symbol.

The productions define a relation $\Rightarrow \subseteq (N \cup T)^* \times (N \cup T)^*$, called *derivations*, on strings: given $L \longrightarrow R$, then $rLt \Rightarrow rRt$ where rLt and rRt are strings in $(N \cup T)^*$. The *language* defined by G is $L(G) = \{q \in T^* \mid s \Rightarrow^* q\}$, where $\Rightarrow^*$ is the transitive, reflexive closure of $\Rightarrow$.

Here $(N \cup T)^*$ and T^* are the strings of symbols and the strings of terminal symbols, respectively.

Given a signature $\Sigma = \langle \mathcal{S}, \mathcal{F} \rangle$ and a set (of nodes) V, we construct a grammar $G_{\Sigma,V}(v)$ for every $v \in V$ as follows. The set of terminals $T = \mathcal{F}$ is the set of function symbols in Σ. For every node $v \in V$ we introduce a new, distinct non-terminal N_v, i.e., $N = \{N_v \mid v \in V\}$ such that $N_v = N_u$ implies $v = u$. To define the productions, we extend Σ to Σ_N (and $\mathcal{F}$ to $\mathcal{F}_N$) by assigning all symbols $N_v \in N$ some (not necessarily distinct) sort s_v from $\mathcal{S}$ and treating the N_v as new constants of the respective sorts. Conversion of a signature to a grammar involves, typically, reversing a function profile $f : S_1 \times \ldots \times S_n \to S$ to a production $S \longrightarrow f(S_1, \ldots, S_n)$. The rules $\longrightarrow_{G_{\Sigma,V}}$ of our grammars are constructed analogously except that, instead of sort symbols we use the constant symbols N_v of the respective sorts. Thus, the productions have the form $N_v \longrightarrow_{G_{\Sigma,V}} t$ where $t \in \mathcal{T}_{\Sigma_N}$ and, for every node $v \in V$, its language $L(G_{\Sigma,V}(v))$ will only contain type correct terms[11]. Letting, finally, for every $v \in V$ the start symbol of its grammar $G_{\Sigma,V}(v)$ be N_v, we obtain the grammar

$$G_{\Sigma,V}(v) = \langle \mathcal{F}, N, \longrightarrow_{G_{\Sigma,V}}, N_v \rangle \tag{2}$$

defining a node-specific language $L(G_{\Sigma,V}(v))$ of terms T_Σ for every node $v \in V$. Denote the collection of such node-specific grammars by $G_{\Sigma,V}$. Define $\overset{G_{\Sigma,V}}{\rightarrowtail}$ such that for all $u, v \in V$ we have

$$u \overset{G_{\Sigma,V}}{\rightarrowtail} v \quad \Leftrightarrow \quad N_v \longrightarrow_{G_{\Sigma,V}} rN_ut \tag{3}$$

for some r, t such that $rN_ut \in \mathcal{T}_{\Sigma_N}$. This gives us a graph structure on V, and we can define a "compatible" *lab*eling function by requiring that

$$\forall v \in V : lab_{G_{\Sigma,V}}(v) \in L(G_{\Sigma,V}(v)) \tag{4}$$

The following fact states the obvious conditions on $G_{\Sigma,V}$ ensuring that (2) and (3) imply uniqueness of the labeling satisfying (4).

Proposition 3.2 *If a collection of node-specific grammars $G_{\Sigma,V}$ (for some signature Σ and a set V) is such that*

1. *for every non-terminal N_v there is exactly one rule $N_v \longrightarrow_{G_{\Sigma,V}} t$, and*
2. *the resulting graph $\Gamma_{G_{\Sigma,V}} = \langle V, \overset{G_{\Sigma,V}}{\rightarrowtail} \rangle$ is well founded,*

then $G_{\Sigma,V}$ determines a unique Σ data dependency.

[11] We can now define a grammar G with a new non-terminal s as start symbol and a rule $s \longrightarrow N_v$ for all $N_v \in N$, i.e., $G = \langle \mathcal{F}, N \cup \{s\}, \longrightarrow_{G_{\Sigma,V}} \cup \{\langle s, N_v \rangle \mid N_v \in N\}, s \rangle$. Then $L(G) = \mathcal{T}_\Sigma$, but we will rather define distinct grammars for each node.

Proof. 2. implies that $\Gamma_{G_{\Sigma,V}}$ is acyclic. That, for every $v \in V$, $L(G(v))$ is a one-element set follows easily: by 1. we have exactly one $N_v \longrightarrow_{G_{\Sigma,V}} t$ for each v, and, since $\Gamma_{G_{\Sigma,V}}$ is well founded, each path from t will uniquely lead to a terminal node. Thus the unique data dependency is $D_{G_{\Sigma,V}} = \langle V, \overset{G_{\Sigma,V}}{\rightarrowtail}, lab_{G_{\Sigma,V}} \rangle$ where $lab_{G_{\Sigma,V}}(v) = q$, where q is the unique string in $L(G_{\Sigma,V}(v))$. $\square$

Another obvious fact following from the assumptions of claim 3.2 is that if $u \overset{G_{\Sigma,V}}{\rightarrowtail} v$ then $lab_{G_{\Sigma,V}}(u)$ is a subterm of $lab_{G_{\Sigma,V}}(v)$.

3.2 Dependencies for recursive functions

The above construction can be used for the special, yet quite frequent and useful, specifications of functions which we call *CR* (for "constructive recursive" [Hav93]). They are defined by recursive equations of special form. Here we will focus our attention on the specific form of (generalised) recurrence relations.

Let $SP = \langle \Sigma, \mathcal{A} \rangle$ be a specification and $S_1 ... S_n, S$ be sorts from Σ. A generalised Σ-term recurrence $R \notin \Sigma$ of type $S_1 \times ... \times S_n \to S$ is given by a set of equations of the form

$$R(\overline{\tau_0}(x)) = \Phi_{\overline{\tau_0}(x)}(\ x, R(\overline{\tau_1}(x)), R(\overline{\tau_2}(x)), \ldots, R(\overline{\tau_z}(x))\ ) \tag{5}$$

where x is a (sequence of) variable(s), for each $0 \leq k \leq z$: $\overline{\tau_k}(x)$ is a sequence of terms $t_{1,k}(x) ... t_{n,k}(x)$ of appropriate sorts, i.e., $t_{i,k}(x) \in \mathcal{T}_\Sigma(x)_{S_i}$, and $\Phi_{\overline{\tau_0}(x)}(x, y_1 ... y_n) \in \mathcal{T}_\Sigma(x, y_1 ... y_n)_S$ with all y_i of sort S. R may be given by a *set* of such equations, i.e., different equations for different (sequences of) term(s) $\overline{\tau_0}(x)$.

From now on we assume that we have chosen a specific algebra A and want to construct a dependency structure for it. As the index set (of nodes) we take the cartesian product $V = S_1^A \times ... \times S_n^A$ of the respective carriers in A. For any assignment $\alpha : x \to A$, let $\alpha_i \in V$ refer to the interpretation of $\overline{\tau_i}(x)$ under α. (I.e., $\alpha_i = \overline{\tau_i}^A(\alpha(x)) = t_{1,i}^A(\alpha(x)) \times ... \times t_{n,i}^A(\alpha(x))$.)

We may now define the set of productions $\longrightarrow_{G_{\Sigma,A}}$ from the recurrence (5) by collecting

$$N_{\alpha_0} \longrightarrow_{G_{\Sigma,A}} \Phi_{\alpha_0}^A(\alpha_0, N_{\alpha_1}, N_{\alpha_2}, \ldots, N_{\alpha_z}) \tag{6}$$

for all assignments α. Gathering the productions $G_{\Sigma,A}(v)$ for every $v \in V$, we obtain a collection of node-specific grammars $G_{\Sigma,A}$ as in (2). The recurrence R from (5) is well-defined in A, i.e., determines a function from V to S^A, if and only if this collection of the node-specific grammars $G_{\Sigma,A}$ satisfies the conditions of Proposition 3.2.

This obviously subsumes primitive recursion for which we have $\Sigma = \langle N, \mathcal{F} \rangle$, the only model A being that of the naturals such that $\mathcal{F}$ are the basic primitive recursive operations on naturals. Specifically, $\Sigma = \langle \{N\}, \{s : N \to N, z : N\} \rangle$, with $N^A = \mathbb{N}$, $s^A = \mathsf{s}$ the successor function, and $z^A = 0$. Given recursive equation schema for an R with $n+1$ variables

$$\begin{aligned} R(0, y_1, \ldots, y_n) &= g(y_1, \ldots, y_n) \\ R(\mathsf{s}(i), y_1, \ldots, y_n) &= h(i, y_1, \ldots, y_n, R(i, y_1, \ldots, y_n)), \end{aligned}$$

where g and h are primitive recursive operations, we get $V = \mathbb{N}^{n+1}$, nodes $y = \langle y_0, y_1, \ldots, y_n \rangle$, and the productions for any fixed values of $y_1 ... y_n$:[12]

$$\begin{aligned} N_{\langle 0, y_1, \ldots, y_n \rangle} &\longrightarrow g(y_1, \ldots, y_n) \\ N_{\langle i+1, y_1, \ldots, y_n \rangle} &\longrightarrow h(i, y_1, \ldots, y_n, N_{\langle i, y_1, \ldots, y_n \rangle}) \end{aligned}$$

[12] Notice that, although we are taking as the set of nodes $\mathbb{N}^{n+1}$, here we would obtain essentially $n+1$ linear graphs (which are isomorphic as graphs, mutually unconnected). The recursion goes only on the first argument and $y_1 \ldots y_n$ function merely as parameters. Thus the only computationally relevant information resides in the first parameter. We do not address here the more specific questions as to how to retrive and utilize this kind of information.

Fibonacci again

As a trivial example, take again the Fibonnacci function from example 2.6.ff. We have the signature $\Sigma = \langle \{N\}, \{s : N \to N, \oplus : N \times N \to N, z : N\} \rangle$, with $R : N \to N$. Taking A to be the natural numbers, we get $V = \mathbb{N}$, $s^A = \mathsf{s}$ the successor function, $\oplus^A = +$, and $z^A = 0$. Applying the above schemata, we obtain the productions for the collection of the node-specific grammars FIB

$$
\begin{aligned}
N_0 &\longrightarrow 1 \\
N_1 &\longrightarrow 1 \\
N_{i+2} &\longrightarrow (N_{i+1} \oplus N_i)
\end{aligned}
$$

The resulting graph and labeling (with parantheses added for readability) are shown in figure 11. Notice the advantage of using this method – we do not obtain the usual tree of recursive calls but a much more efficient graph Γ_{FIB}.

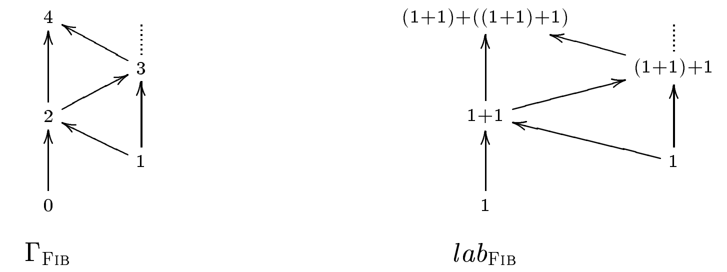


Figure 11: The graph and labeling resulting from the collection of node-specific grammars FIB.

3.3 Parallelisation - Fast Fourier Transform

To illustrate the appliction to actual parallelisation of programs, we review the technique of fast Fourier Transform, FFT, due to [CT65]. We do not focus here on the development process, but merely recast an existing solution of the problem in our formalism: we explain briefly the background, apply a designed data dependency and map it onto a hypercube. In the last paragraph we show how this solution can be obtained directly from the defining equations using the technique from 3.1-3.2.

The numerical problem can be summarised as follows: In many areas, computing of the n-order polynomials $p(x) = \sum_{i=0}^{n} p_i x^i$, $0 \leq n$, for certain argument values $x = x_1, \ldots, x_k$ is important. These areas include, for instance, signal processing, where evaluation of the polynomials for the *complex roots of unity* plays a central role.

An n'th complex root of unity is a number ω such that $\omega^n = 1$. For every $n \geq 1$ there are n distinct complex roots, denoted ω_n^j, and satisfying

$$
\begin{aligned}
\omega_n^0 &= 1 \\
\omega_n^{j+1} &= \omega_n^j \omega_n^1 \\
\omega_n^j &= \omega_n^{j \bmod n}
\end{aligned}
$$

(The second equation is trivial for $j = 0$.) Useful facts about these roots are that $(\omega_{2n}^j)^2 = \omega_{2n}^{2j} = \omega_n^j$ and $\omega_{2n}^{j+n} = -\omega_{2n}^j$. The roots may be computed by $\omega_n^j = e^{2\pi i/n} = cos(2\pi j/n) + sin(2\pi j/n)i$, where $i = \omega_4^1$ is the imaginary unit, i.e., $i^2 = -1$.

Cooley and Tukey [CT65] discovered a very efficient fast Fourier transform for computing an n-order polynomial for all n'th complex roots of unity. Let us consider this method for the case where $n = 2^M$, i.e., the one-dimensional binary version. Then $p(x) = p_e(x^2) + x * p_o(x^2)$, where

$p_e(x) = \sum_{i=0}^{2^{n-1}} p_{2i} * x^i$ and $p_o(x) = \sum_{i=0}^{2^{n-1}} p_{2i+1} * x^i$. Taking into account the special properties of the roots of unity, we get

$$\begin{aligned} p(\omega_{2^m}^j) &= p_e(\omega_{2^{m-1}}^j) + \omega_{2^m}^j p_o(\omega_{2^{m-1}}^j) \\ p_e(\omega_{2^{m-1}}^j) &= p_{e,e}(\omega_{2^{m-2}}^j) + \omega_{2^{m-1}}^j p_{e,o}(\omega_{2^{m-2}}^j) \\ p_o(\omega_{2^{m-1}}^j) &= p_{o,e}(\omega_{2^{m-2}}^j) + \omega_{2^{m-1}}^j p_{o,o}(\omega_{2^{m-2}}^j) \end{aligned}$$

This can be formulated as a recurrence equation

$$\begin{aligned} p(\omega_{2^M}^j) &= F(j, M) \\ F(j, m+1) &= F(s_0(j, m+1), m) + \omega_{2^{m+1}}^j F(s_1(j, m+1), m) \\ F(j, 0) &= p_j, \end{aligned} \tag{7}$$

where $s_0(j,m) = j - (j \bmod 2^m) + (j \bmod 2^{m-1})$ sets the m'th bit of j to 0 and $s_1(j,m) = j - (j \bmod 2^m) + 2^m + (j \bmod 2^{m-1})$ sets the m'th bit of j to 1. Bits are the 0's and 1's in the binary representation of a number, and are counted with the least significant bit as number 1.

The stencil of the recurrence F – the *butterfly* graph – is illustrated in figure 12 for $M = 3$.

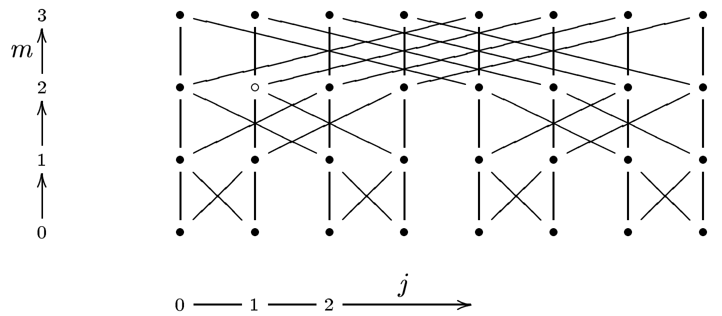


Figure 12: The butterfly pattern of the recurrence F.

Columns correspond to the j and rows to the m arguments with the leftmost column and bottom row numbered 0. Thus, for instance, the value of $F(2,1)$ resides at the row 2 column 1 (marked with the circle instead of bullet on the drawing). Increasing M by one, i.e., taking $n = 2^{M+1}$, amounts to putting two butterflies of height M side by side and connecting uppermost nodes (in the M'th row) of the one butterfly to the respective nodes of the other in the row $M+1$.

Now, figure 13 shows the topology of hypercubes of dimension $M = 3$ and 4. Each of the 2^M processors has M bidirectional communication channels. A hypercube of dimension $M+1$ is obtained from two hypercubes of dimension M by connecting each node of one hypercube to corresponding node of the other. The figure 4.1 shows the embedding of the butterfly dependency

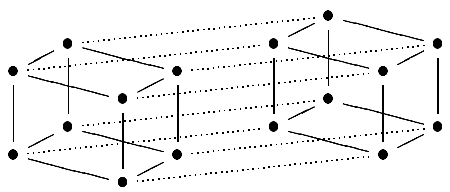


Figure 13: Connections for hypercubes of dimension 3 (solid), and 4 (solid and dotted).

of height 3 from figure 12 into the space-time representation of the hypercube of dimension 3 from figure 13 (due to the "`exceeded TeX capacity`", the figure has been divided into two parts 14 and 14.b on page 17.) Using the regularity in building a butterfly and a hypercube of a higher dimension, an analogous embedding is obtained for a butterfly of arbitrary height M into a hypercube of dimension M.

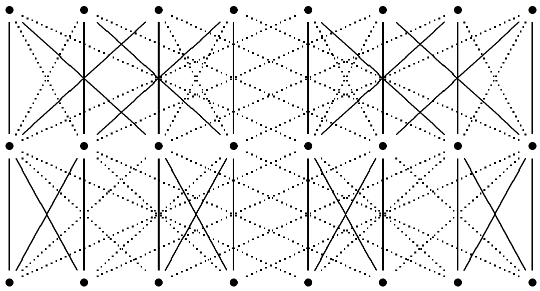


Figure 14: The rows 0-2 of the butterfly implemented on a hypercube.

Construction of dependency for FFT

Applying the strategy from 3.1-3.2, we obtain the same dependency structure directly from the definition. The equations (7) define a recurrence relation in two arguments. Working again with the algebra of natural numbers, we take as the set of nodes V the cartesian product $\{\langle i,j\rangle \in \mathbb{N}\times\mathbb{N} \mid 0 \leq i < 2^M, 0 \leq j \leq M\}$. The grammar resulting from the (last two) equations of (7) is then:

$$\begin{array}{rcl} N_{\langle j,0\rangle} & \longrightarrow & p_j \\ N_{\langle j,m+1\rangle} & \longrightarrow & N_{\langle s_0(j,m+1),m\rangle} \oplus \omega^j_{2^{m+1}} N_{\langle s_1(j,m+1),m\rangle} \end{array}$$

with the coefficients p_j of the polynomial treated as input since the roots of unity are fixed once M is known. The resulting shape of the data dependency is precisely the one given in figure 12. Again, observe how this method leads directly to an "appropriate" graph instead of generating the full tree of recursive calls.

4 Variations and Remarks

The framework of communication algebras introduced in section 2 is only a very general setting. A series of issues, concerning both refinment of the general concepts and their application in a concrete context, has not been addressed and demands further research. In this section we make just a few observations concerning such aspects as well as possible extensions. We begin by listing some additional properties of the category of communication algebras not mentioned earlier.

4.1 Some general facts about $\mathsf{CAlg}(\mathsf{A},\mathsf{D})$.

1. Double category.

The following proposition allows us to view $\mathsf{CAlg}(\mathsf{A},\mathsf{D})$ as a double category with horizontal and vertical arrows, respectively, of the form $\langle\vec{h}, id\rangle$ and $\langle id, \bar{h}\rangle$. That is, the horizontal part is essentially D and the vertical part A. The nice implication of this is that we can work relatively independently with the algebra- and the communication-part of the objects in $\mathsf{CAlg}(\mathsf{A},\mathsf{D})$.

Proposition 4.1 *Each morphism $h : C \to D$ in $\mathsf{CAlg}(\mathsf{A},\mathsf{D})$ can be factored so that*

1. $h = \langle\bar{h}, id_{\vec{C}}\rangle \circ \langle id_{\bar{D}}, \vec{h}\rangle$,

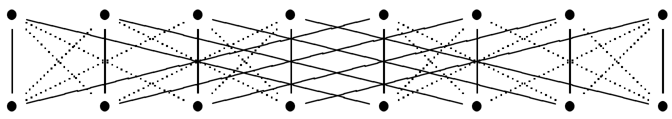
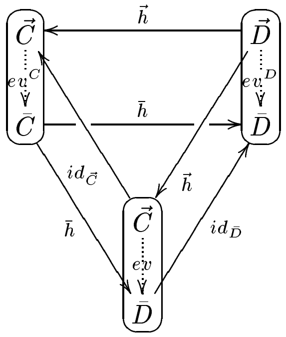


Figure 14.b:Embedding of the rows 2-3 of the butterfly into a hypercube.

2. $h = \langle id_{\bar{C}}, \vec{h} \rangle \circ \langle \bar{h}, id_{\vec{D}} \rangle$.

Proof. The following diagram details the first case:



The intermediary C-algebra is as shown in the figure with $ev = ev^C \circ \bar{h}$. This makes $\langle \bar{h}, id_{\vec{C}} \rangle$ a C-homomorphism $\langle \bar{C}, \vec{C}, ev^C \rangle \to \langle \bar{D}, \vec{C}, ev \rangle$. Also, since h is a C-homomorphism, we have for all $i \in I^{\vec{D}} : \bar{h}(ev^C(\vec{h}(i))) = ev^D(i)$, $\vec{h} \circ ev^C \circ \bar{h} = ev^D$, $id_{\bar{D}}(ev(\vec{h}(i))) = ev^D(i)$, i.e., $\langle id_{\bar{D}}, \vec{h} \rangle$ is a C-homomorphism $\langle \bar{D}, \vec{C}, ev \rangle \to \langle \bar{D}, \vec{D}, ev^D \rangle$.

The second case is entirely analogous with the intermediary algebra being $\langle \bar{C}, \vec{D}, \vec{h} \circ ev^C \rangle$. $\square$

2. Adjunction.

Another general fact is the following:

Proposition 4.2 *If* A *has an initial object* T_{A}*, then the functor* $F : \mathsf{D}^{op} \to \mathsf{CAlg}(\mathsf{A}, \mathsf{D})$ *defined by* $F(\vec{C}) = \langle T_{\mathsf{A}}, \vec{C} \rangle$ *and* $F(\vec{h}) = \langle id_{T_{\mathsf{A}}}, \vec{h} \rangle$ *is left adjoint to Dep,* $F \dashv Dep$*, with identity as unit.*

Thus, if D^{op} has an initial object, so does $\mathsf{CAlg}(\mathsf{A}, \mathsf{D})$. However, in general, this won't be the case since interesting subcategories of $\mathsf{DD}(\Sigma)$ do not have terminal objects (D^{op} do not have initial objects). In particular $\emptyset$ is initial in Gr^+ and in $\mathsf{DD}(\Sigma)$, but there is no terminal objects there. (If Z_{A} is terminal in A, then $\langle Z_{\mathsf{A}}, \emptyset \rangle$ is terminal in $\mathsf{CAlg}(\mathsf{A}, \mathsf{DD}(\Sigma))$.)

We do not focus on the notion of initiality and the relevance of empty communication part is probably limited to the fact that it allows us to treat usual algebras as communication algebras with an empty dependency.

3. Refinement.

Definition 2.8 has following implications for the notion of refinement. If an (A, D)-morphism $h : C \to D$ is taken as a model of the refinement notion for communication algebras, we obtain the standard refinement of algebras $\bar{h} : \bar{C} \to \bar{D}$. Refinement of dependencies, however, is the contravariant Gr^+- mapping $\vec{h} : Gr(\vec{D}) \to Gr(\vec{C})$. That is, all the dependencies from $\vec{D}$ have to be preserved, but they may be "streched" and augmented with new dependencies, as the example in figure 15 illustrates. We let $\vec{h}(a) = a$, $\vec{h}(b) = d$, and $\vec{h}(f) = f$. The commutativity requirement, $\bar{h}(ev^C(\vec{h}(i))) = ev^D(i)$, restricts the legal combinations of $\vec{h}$ and $\bar{h}$. In particular, the labels in $\vec{C}$ may differ from those in $\vec{D}$ like with $\vec{h}(b) = d$, only if d evaluates in $\bar{C}$ to a value x representing b^D, i.e., such that $\bar{h}(x) = b^D$.

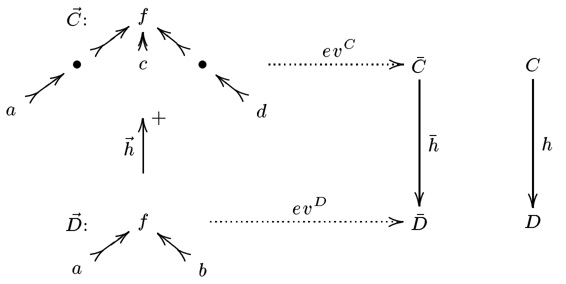


Figure 15: An example of refinement of communication algebras.

Also, refinement "does not care" about what is computed in C at the nodes which are not in the image of $\vec{h}(\vec{D})$. There may be new nodes and dependencies, also on nodes labelled by terms not occurring in the labelling of $\vec{D}$ (like $c \rightarrowtail f$ in $\vec{C}$). In particular, if we admit a communication algebra with an empty dependency part (i.e., empty carriers in the category D), a standard algebra $\bar{A}$ may be considered a communication algebra with empty communication part, $A = \langle \bar{A}, \emptyset \rangle$. Then any algebra B with non-empty communication part will be a refinement of A provided there is a homomorphism $\bar{h} : \bar{B} \to \bar{A}$.

As an example of possible drawback, consider an algebra D with the sorts S, S_1, S_2, and the dependency $s \rightarrowtail f(s)$ for two terms of sort S. Furthermore, let the sort S be the Cartesian product of the two other sorts $S = S_1 \times S_2$, with the obvious projections $\pi_i : S \to S_i$ and pairing function $\langle \_, \_ \rangle : S_1 \times S_2 \to S$. Let $\bar{D}$ satisfy the natural equalities for these functions and, in addition, let $\bar{D} \models s = \langle s_1, s_2 \rangle$, and $\bar{D} \models f(s) = \langle f_1(s_1), f_2(s_2) \rangle$. This allows us to design the refinement of the dependency structure illustrated in figure 16 which together with the identity on $\bar{D}$ yields a C-homomorphism $\langle id_{\bar{D}}, \vec{h} \rangle$.

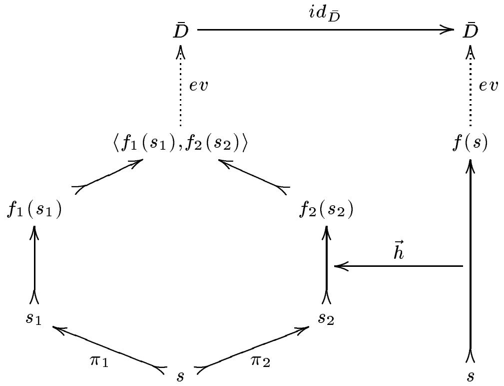


Figure 16: Another example of refinement of communication algebras.

One may say that it is counterintuitive – the dependency $s \rightarrowtail f(s)$ in $\vec{D}$ is refined to *both* paths (through s_1 and s_2). Our morphism, however, sends the former only on one of them. This might indicate the need for a more specific definition of dependency morphism. For the time being we have settled for the one which seems simple enough – in this case it is better to have a morphism like the one above rather than not having any. Most alternatives seem to prevent one from designing a morphism at least in some cases where one would expect (like to have) one.

4.2 Communication Algebras over Different Signatures

So far we have only considered C-algebras over one given signature. Thus, the development process illustrated in figure 9 is not quite realistic since it is rather certain that such a process will involve change of signatures. In this subsection we extend the standard way of flattening the categories of algebras indexed by signatures to the C-algebras and their implementations. Thus we obtain the means of refining the picture of the development process from figure 9 by also admitting transitions between C-algebras over different signatures.

As announced in subsection 2.1, we let Sig be the category of signatures with injective signature morphisms.[13]

Definition 4.3 Let $\sigma : \Sigma' \to \Sigma \in Mor(\mathsf{Sig})$ and $D \in \mathsf{DD}(\Sigma)$. The *$\sigma$-reduct* of D, $D|_\sigma \in \mathsf{DD}(\Sigma')$, is defined in two steps as follows:

1. let X be D with restricted re-labelling : $I^X = I^D$, $\overset{X}{\rightarrowtail} = \overset{D}{\rightarrowtail}$ and $lab^X(i) = \{t : \sigma(t) = lab^D(i)\}$;
2. $D|_\sigma$ is then given by
   - $I^{D|_\sigma} = I^D \setminus \{i : lab^X(i) = \emptyset\}$,
   - $lab^{D|_\sigma}(i) = t \in lab^X(i)$, and
   - $i \overset{D|_\sigma}{\rightarrowtail} j \Leftrightarrow \begin{cases} i \overset{D}{\rightarrowtail} j \quad or \\ \exists\, i \overset{X}{\rightarrowtail} k_1 \cdots k_z \overset{X}{\rightarrowtail} j : 1 \leq n \leq z \Rightarrow lab^X(k_n) = \emptyset \end{cases}$

In the first step, we remove all the labels from D which are not in the image of σ, and re-label the remaining ones to their σ pre-image. In the second we convert paths with unlabelled intermediary nodes into edges between the labelled end-points.

For $A \in \mathsf{Alg}(\Sigma)$ and $\sigma : \Sigma' \to \Sigma \in Mor(\mathsf{Sig})$, we let $A|_\sigma$ denote the usual reduct. We then have the extension of the standard result:

Lemma 4.4 *$\sigma : \Sigma' \to \Sigma \in Mor(\mathsf{Sig})$ induces the functor $\_|_\sigma : \mathsf{CAlg}(\Sigma) \to \mathsf{CAlg}(\Sigma')$.*

Proof. For an $A \in \mathsf{CAlg}(\Sigma)$, define $A|_\sigma = \langle \bar{A}|_\sigma, \vec{A}|_\sigma \rangle$. For a morphism $h : A \to B \in Mor(\mathsf{CAlg}(\Sigma))$, $h|_\sigma = \langle \bar{h}|_\sigma, \vec{h}|_\sigma \rangle$ where $\bar{h}|_\sigma$ is the image of $\bar{h}$ under the usual reduct functor for algebras, while $\vec{h}|_\sigma$ is the restriction of $\vec{h}$ to $\vec{B}|_\sigma$ under changed labelling: each node i of $\vec{B}|_\sigma$ is sent to the node $\vec{h}(i)|_\sigma$, and edge e of $\vec{B}|_\sigma$ onto the edge (path) $\vec{h}(e)|_\sigma$ in $\vec{A}|_\sigma$. This is a well defined $\mathsf{DD}(\Sigma')$-morphism: commutativity $\vec{h} \circ ev^A \circ \bar{h} = ev^B$ implies the corresponding commutativity in the reduct, since the latter operations are restrictions of their pre-images. $\square$

Thus $\_|\_ : \mathsf{Sig}^{op} \to \mathsf{CAT}$ sending Σ to $\mathsf{CAlg}(\Sigma)$ and σ to $\_|_\sigma$ is an indexed category. By the standard application of the Grothendieck construction we may define the flattened category CAlg with 1) objects $\langle \Sigma, A \rangle$ where $A \in Obj(\mathsf{CAlg}(\Sigma))$, 2) morphisms of the form $\langle \sigma, f \rangle : \langle \Sigma', A \rangle \to \langle \Sigma, B \rangle$, where $\sigma \in Mor(\mathsf{Sig})$ and $f : A \to B|_\sigma \in Mor(\mathsf{CAlg}(\Sigma'))$, and 3) composition $\langle \sigma, f \rangle \circ \langle \rho, g \rangle = \langle \sigma \circ \rho, f \circ g|_\rho \rangle$.

Quite an analogous construction yields an indexed category $\_\|\_ : \mathsf{Sig}^{op} \to \mathsf{CAT}$ which sends Σ to $\mathsf{Imp}(\Sigma)$. For $\sigma : \Sigma' \to \Sigma$, the induced functor $\_\|_\sigma : \mathsf{Imp}(\Sigma') \to \mathsf{Imp}(\Sigma)$, sends an implementation $f_{A \to W} \in \mathsf{Imp}(\Sigma)$ onto $(f\|_\sigma)_{A|_\sigma \to W} \in \mathsf{Imp}(\Sigma')$, where $f\|_\sigma$ is the restriction of f to $\vec{A}|_\sigma$. The result of the flattening is the category Imp. We do not dwell on this notion, because we consider the category $\mathsf{Imp}(\Sigma)$ as more central – porting *implementations* does not typically involve changes of signature.

4.3 More specific dependency morphisms

The class $\mathsf{CAlg}(\Sigma)$, although a legitimate entity in our setting, is of little interest. The reason is that the arbitrary dependency morphisms in $\mathsf{DD}(\Sigma)$ make the connection between the algebra- and the communication-part of the objects in $\mathsf{CAlg}(\Sigma)$ rather loose and indefinite. We mention two possible restrictions.

[13] We restrict ourselves to the injective signature morphisms for one main reason: defining a reduct functor over data dependencies for non-injective morphisms implies several choices which seem slightly arbitrary. We therefore prefer to postpone such a decision until we have more experience and reasons to choose one alternative rather than another.

4.3.1 C-Algebras with Compatible D-morphisms

Let A be a fixed class of Σ-algebras. It is natural to think of it as a model class of some (equational) specification. One may postulate that such a choice of A should restrict the relevant dependency morphisms. In particular, if $\mathsf{A} \not\models s = t$, the D-morphisms should not be allowed to map nodes labelled by s to ones labelled by t (nor vice versa), since such mappings are not compatible with the algebras in A. Thus, compatibility of a $\mathsf{DD}(\Sigma)$ morphism $\vec{h} : \vec{C} \to \vec{D}$ wrt. an algebra A means that $\vec{h}$ does not effect a re-labelling which is inconsistent with valuation in A. More precisely:

Definition 4.5 Let $\vec{h} : \vec{C} \to \vec{D} \in \mathit{Mor}(\mathsf{DD}(\Sigma))$ and $A \in \mathsf{A} \subseteq \mathsf{Alg}(\Sigma)$.

1. $\vec{h}$ is *A-compatible* iff $\forall i \in I^{\vec{C}} : (lab^{\vec{D}}(\vec{h}(i)))^A = (lab^{\vec{C}}(i))^A$;
2. $\vec{h}$ is *A-compatible* iff it is A-compatible for all $A \in \mathit{Obj}(\mathsf{A})$;
3. $\mathsf{DD}(\mathsf{A})$ is a wide subcategory of $\mathsf{DD}(\Sigma)$ with A-compatible morphisms.

Alternatively, $\vec{h}$ is A-compatible iff for each $A \in \mathit{Obj}(\mathsf{A})$, $\langle id_A, \vec{h}\rangle$ is a C-homomorphism $\langle A, \vec{D}\rangle \to \langle A, \vec{C}\rangle$.

We have the obvious

Proposition 4.6 *Suppose that A has an initial object T_{A}, and let $\vec{h} : \vec{C} \to \vec{D} \in \mathit{Mor}(\mathsf{DD}(\Sigma))$. $\vec{h}$ is A-compatible $\Leftrightarrow$ $\vec{h}$ is T_{A}-compatible.*

Proof. ($\Rightarrow$) is obvious. For ($\Leftarrow$) assume $\vec{h}$ is T_{A}-compatible. This means that the lower triangle of the following diagram commutes, where ev assigns the values in the algebra to the nodes according to their labels.

$$\begin{tikzcd}
 & A & \\
 & T_{\mathsf{A}} \arrow[u, "a"] & \\
\vec{C} \arrow[uur, "ev^A"] \arrow[ur, "ev"] \arrow[rr, "\vec{h}"'] & \circlearrowleft & \vec{D} \arrow[uul, "ev^A"'] \arrow[ul, "ev"']
\end{tikzcd}$$

Since T_{A} is initial, for each $A \in \mathsf{A}$, $ev^A = ev \circ a$ where a is the unique morphism $T_{\mathsf{A}} \to A$. But then $\vec{h} \circ ev = ev \Rightarrow (\vec{h} \circ ev) \circ a = ev \circ a \Rightarrow \vec{h} \circ ev^A = ev^A$. $\square$

As a special case, the morphisms compatible with $\mathsf{A} = \mathsf{Alg}(\Sigma)$ are the ones which do not change the labelling at all, since they have to be compatible with the initial word algebra T_Σ.

For communication algebras with compatible dependency morphism, we also have:

Lemma 4.7 *$Dep : \mathsf{CAlg}(\mathsf{A}, \mathsf{DD}(\mathsf{A})) \to \mathsf{DD}(\mathsf{A})^{op}$ is a split fibration.*

Proof. We use the notation from the diagram to the right which illustrates the fibration situation in the present context (the dashed arrows indicate the directions in $\_^{op}$ categories):

$$\begin{tikzcd}
\mathsf{CAlg}(\mathsf{A}, \mathsf{DD}(\mathsf{A})) \arrow[d, "Dep"] \\
\mathsf{DD}(\mathsf{A})^{op}
\end{tikzcd}
\qquad
\begin{tikzcd}
B \arrow[d, "f"'] \arrow[dr, "g"] & \\
C \arrow[r, "h"] & D \\
\vec{C} \arrow[r, dashed, "\vec{h}"] & \vec{D}
\end{tikzcd}
\qquad
\begin{tikzcd}
\bar{B} \arrow[dr, "\bar{g}"'] \arrow[drr, "\bar{g}"] & & \\
 & \bar{D} \arrow[r, "id_{\bar{D}}"] & \bar{D} \\
 & \vec{C} \arrow[u, dotted, "ev^C"] \arrow[uul, dotted, "ev^B"] & \vec{D} \arrow[u, dotted, "ev^D"'] \arrow[l, "\vec{h}"']
\end{tikzcd}$$

For a $\mathsf{DD}(\mathsf{A})^{op}$-morphism $\vec{h} : \vec{C} \dashrightarrow \vec{D}$ and $D = \langle \bar{D}, \vec{D}, ev^D\rangle$, the cleavage is defined as

$$cl(\vec{h}, D) \overset{\text{def}}{=} h : C \to D \ \mathit{where} \begin{cases} h = \langle \vec{h}, id_{\bar{D}}\rangle \\ C = \langle \bar{D}, \vec{C}, ev^C\rangle, \end{cases} \tag{8}$$

where ev^C is induced by $lab^{\vec{C}}$ according to definition 2.7.3. Since $\vec{h}$ is A-compatible, $\vec{h} \circ ev^C \circ id_{\bar{D}} = ev^D$, and so h is a C-homomorphism.

If $g : B \to D$ is such that $Dep(g) = \vec{h}$, the unique $f : B \to C$ with $Dep(f) = id_{\vec{C}}$ making $f \circ h = g$ is $\langle \bar{g}, id_{\vec{C}} \rangle$ (i.e., $\langle \bar{f}, id_{\vec{C}} \rangle \circ \langle id_{\bar{D}}, \vec{h} \rangle = \langle \bar{g}, \vec{h} \rangle$).

Obviously, the cleavages compose: for $\vec{h} : \vec{D} \to \vec{C}$, $\vec{g} : \vec{C} \to \vec{F}$ (i.e., $\vec{g} : \vec{F} \dashrightarrow \vec{C}$ and $\vec{h} : \vec{C} \dashrightarrow \vec{D}$) and $\langle \bar{D}, \vec{D} \rangle \in Dep^{-1}(\vec{D})$ we have $cl(\vec{h}, D) = \langle \vec{h}, id_{\bar{D}} \rangle$, $cl(\vec{g}, \langle \vec{C}, \bar{D} \rangle) = \langle \vec{g}, id_{\bar{D}} \rangle$, so $cl(\vec{g}, \langle \vec{C}, \bar{D} \rangle) \circ cl(\vec{h}, \langle \vec{D}, \bar{D} \rangle) = \langle \vec{h} \circ \vec{g}, id_{\bar{D}} \rangle = cl(\vec{h} \circ \vec{g}, D)$. Hence Dep is a fibration.

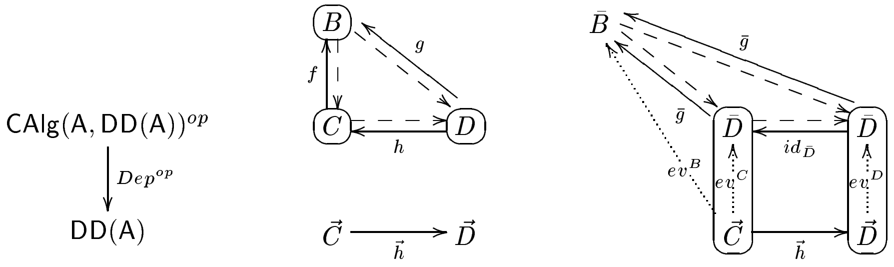
It is split: $(id_{\vec{D}})^*(\langle \bar{D}, \vec{D} \rangle) = \langle \bar{D}, \vec{D} \rangle$, so the natural transformation $id_{Dep^{-1}(\vec{D})} \rightrightarrows (id_{\vec{D}})^*$ is identity. Also, since cleavages compose : $(\vec{g})^* \circ (\vec{h})^* = (\vec{h} \circ \vec{g})^*$. $\square$

The fiber over a given data dependency corresponds here to all the communication algebras with the same operations of interest which are to be evaluated in the way prescribed by the dependency structure.

For the compatible dependency morphisms, we also have the additional fact:

Lemma 4.8 *$Dep : \mathsf{CAlg}(\mathsf{A}, \mathsf{DD}(\mathsf{A})) \to \mathsf{DD}(\mathsf{A})^{op}$ is a cofibration.*

Proof. The proof and definition of cocleavage are essentially the same as in the lemma 4.7. We have the following picture:



For a morphism $\vec{h} : \vec{C} \to \vec{D}$ and $D = \langle \bar{D}, \vec{D}, ev^D \rangle$, the cocleavage is defined as

$$ccl(\vec{h}, D) \stackrel{\text{def}}{=} \langle id_{\bar{D}}, \vec{h} \rangle : D \dashrightarrow \langle \bar{D}, \vec{C}, ev^C \rangle \tag{9}$$

with ev^C given by the last point of definition 2.7. Since $\vec{h}$ is A-compatible, the cocleavage is a morphism in $\mathsf{CAlg}(\mathsf{A}, \mathsf{DD}(\mathsf{A}))$. Trivial repetition of the arguments from the proof of theorem 4.7 yields the conclusion. $\square$

This, together with the lemma 4.7, means that Dep is a bifibration and gives

Corollary 4.9 *Reindexing functors induced by (Dep, cl) have left adjoints.*

Referring back to subsection 4.2, it is also easy to see that, for an injective signature morphism $\sigma : \Sigma' \to \Sigma$, if $\vec{h} : \vec{C} \to \vec{D} \in Mor(\mathsf{DD}(\Sigma))$ is $\bar{A}$-compatible for an $\bar{A} \in Obj(\mathsf{Alg}(\Sigma))$, then $\vec{h}|_\sigma$ is $\bar{A}|_\sigma$-compatible. Hence, the image under $\_|_\sigma$ of a $\mathsf{CAlg}(\mathsf{A}, \mathsf{DD}(\mathsf{A}))$ is a category $\mathsf{CAlg}(\mathsf{A}|_\sigma, \mathsf{D})$ where $Obj(\mathsf{A}|_\sigma) = \{A|_\sigma : A \in Obj(\mathsf{A})\}$ and morphisms in D are $\bar{\mathsf{A}}|_\sigma$-compatible.

4.3.2 Graph-morphisms

We believe that refinement of data dependencies – at least as expressed by the compatible dependency morphismsin in def. 4.5 – may be a practical notion for the development of implementations and, perhaps, more abstract algorithms. Nevertheless, one could argue that it is not needed in such a generality since data dependencies are quite concrete objects introduced towards the very end of the development process. In particular, passing from one communication algebra to another should not imply the introduction of new dependencies, only (and at most) purely syntactic memoisation. It is only the implementation morphisms which adjust the dependency graph to match it to the communication structure of a given architecture. Accepting this view, one would work with a class of Σ-algebras $\mathsf{A} \subseteq \mathsf{CAlg}(\Sigma)$, and with the dependency morphisms which are simply Gr-morphisms (special case of Gr^{+}-morphisms) compatible with $\mathsf{CAlg}(\Sigma)$. Let us denote this class of C-algebras by $(\mathsf{A}, \mathsf{D}^{\mathsf{Gr}}_{\Sigma})$. (Obviously, it is a subcategory of $(\mathsf{A}, \mathsf{DD}(\mathsf{A}))$.) $\mathsf{D}^{\mathsf{Gr}}_{\Sigma}$-morphisms do not perform any relabelling whatsoever ($\vec{h} : \vec{D} \to \vec{C}$ can only identify some nodes from $I^{\vec{D}}$ with identical labels), so in this class of C-algebras, any pair $\vec{h} \in Mor(\mathsf{D}^{\mathsf{Gr}}_{\Sigma})$ and $\bar{g} \in Mor(\mathsf{A})$ will give rise to an $(\mathsf{A}, \mathsf{D}^{\mathsf{Gr}}_{\Sigma})$-morphism.

Since $Mor(\mathsf{D}^{\mathsf{Gr}}_{\Sigma}) \subset Mor(\mathsf{DD}(\mathsf{A}))$, we can trivially repeat the constructions from lemmas 4.7 and 4.8, to see that $Dep : (\mathsf{A}, \mathsf{D}^{\mathsf{Gr}}_{\Sigma}) \to (\mathsf{D}^{\mathsf{Gr}}_{\Sigma})^{op}$ is a bifibration.

The class $(\mathsf{A}, \mathsf{D}^{\mathsf{Gr}}_{\Sigma})$, although theoretically perhaps not the most fascinating one, deserves to be mentioned because it was used in practical applications and underlies the design of the specification formalism SAPPHIRE [Hav93, ČH95].

4.4 Nondeterminism

Since parallelism often is a source of nondeterminism, we suggest here how we can incorporate that phenomenon into the present framework as well. There are at least two ways to do it. The first, 4.4.1, utilizes the notion of multialgebras, which are designed for incorporating nondeterminism into algebraic specification as an abstraction mechanism [WM94]. The second, 4.4.2, is biased towards the operational view, according to which nondeterminism is something which arises only during the actual computations.

4.4.1 The Multialgebraic View

As mentioned earlier, the category of algebras used in communication algebras is not necessarily fixed once and for all. Instead of the usual Σ-algebras, we can work with a different category of $\mathsf{Alg}(\Sigma)$. For modelling nondeterminism, a natural choice is (some) category of multialgebras with multihomomorphisms, [Hus93], [WM94]. A multialgebra is an algebra where operations applied to single arguments may return *sets* of elements, corresponding to the possible results of a nondeterministic operation. Composition is defined pointwise. The notion of homomorphism $h : A \to B$ is generalised so that one requires $h(f(x)^A) \subseteq f^B(h(x))$. A (ground) term is interpreted in a multialgebra as the set (of its possible results). Thus, in the language one can use nondeterministic terms with the paradigmatic example being nondeterministic binary choice $\_ \sqcup \_ : S \times S \to S$, for some sort S. [14]

The change of the category of algebras is the only modification needed. The definition of C-homomorphism h (with $\bar{h}$ being a multihomomorphism) will allow us, for instance, to implement an algebra B with a more deterministic algebra A as illustrated in figure 17. All operations except $\sqcup$ are deterministic. The possibility of $\vec{h}$ relabelling $s \sqcup t$ to s follows here directly from the definition of multihomomorphism since, for $i \in I^B$ with $lab(i) = s \sqcup t$, we have that $\bar{h}(ev^A(\vec{h}(i))) = \bar{h}(s^A) = s^B \subseteq s^B \cup t^B = (s \sqcup t)^B = ev^B(i)$.

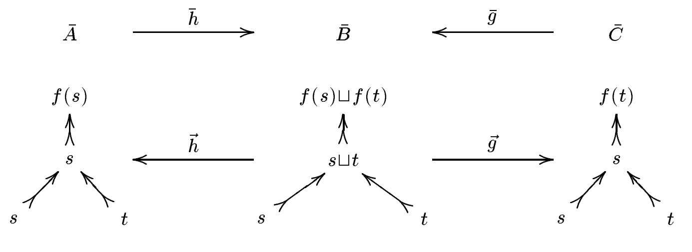


Figure 17:

[14] There exist different specification languages which all share the common primitive predicate, $s \prec t$, interpreted as t being at least as nondeterministic as s.

4.4.2 The Operational View

One source of nondeterminism is purely operational – the differences in relative speed of various processors on various machines being unpredictable. A paradigmatic example of this is a processor P waiting for an input from one of several other processors. P processes the data which arrives first – without discriminating against any of the sources – and discards any later arrivals. Multialgebras provide a more abstract than such a low-level view of nondeterminism. Here we sketch a possibility for modelling also such a low-level view.

The computation related information for a C-algebra $A = \langle \bar{A}, \vec{A} \rangle$ is represented by its communication part. This allows us to retain the standard notion of algebra $\bar{A}$ and to model nondeterminism in $\vec{A}$. We simply allow for labelling the nodes of $\vec{A}$ by *sets of terms*, rather then unique terms. The set labelling a node represents the possible values computed at the node. The source of the evaluation function ev^A is then $\bigcup_{i \in I^{\vec{A}}} \{\langle i, t \rangle : t \in lab^A(i)\}$. The nodes act as oracles – in an actual computation each one will produce a unique value. But this value is not known in advance, and so the labelling and the ev function merely indicate the range of possibilities.

Definition 4.10 $\mathsf{DD}^n(\Sigma)$ is the category where (Γ is as in definition 2.1):

1. objects are Σ n-data dependencies – algebras over the signature $\Gamma^n_\Sigma = \Gamma \cup \{lab : I \to \mathcal{P}^+(\mathcal{T}_\Sigma)\}$, where $\forall i\ \forall s, t \in lab(i) : Sort(s) = Sort(t)$;
2. a $\mathsf{DD}^n(\Sigma)$-morphism $\vec{h} : B \to A$ is a pair $\langle \vec{h}, \vec{h_l} \rangle$ where
   - $\vec{h}$ is a (sort compatible) Gr^+-morphism $\vec{h} : Gr(B) \to Gr(A)$,
   - $\vec{h_l}$ is the family $\langle h_i : lab^A(\vec{h}(i)) \to lab^B(i) \rangle_{i \in I^B}$.

Sort compatibility means that the morphisms have to respect the sorting of labels: if labels of i are of sort S, then so must be the labels of $\vec{h}(i)$. The $\vec{h_l}$ mapping picks the subset of labels of $lab^B(i)$ which is the pre-image of the label set $lab^A(\vec{h}(i))$. For instance, in figure 18, $\vec{B}$ is a dependency for nondeterministic choice between s and t which is then propagated as the argument to the function f. (The relabelling is merely a restriction and $\vec{h_l}$ is the obvious inclusion.)

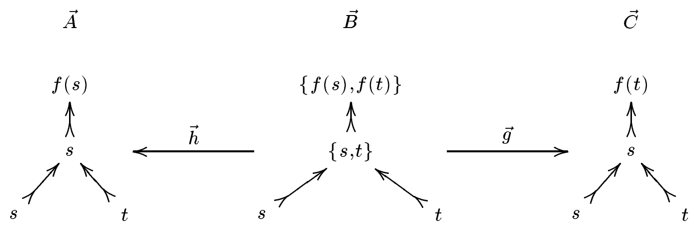


Figure 18:

The node i in $\vec{B}$ with the label $\{s, t\}$ may compute s or t. Similarly, the topmost node j may compute $f(s)$ or $f(t)$. The dependency $i \rightarrowtail j$ says that when i computes one of s or t, j will compute one of $f(s)$ or $f(t)$. A possible determinisitic refinement of $\vec{B}$ which always chooses s and computes only $f(s)$ is given in $\vec{A}$ with the morphism $\vec{h}$.

Observe, however, that the definitions of an n-dependency and a $\mathsf{DD}^n(\Sigma)$-morphism do not take into account the possible semantic information about *how* the computation at j depends on the result delivered by i. In principle, it admits the situations like C, where we obtain $s \rightarrowtail f(t)$. (The same occurred in figure 17.) The morphism only captures the preservation of dependencies (in terms of the underlying graphs) and the nonincreasing nondeterminism. That the computation of $f(t)$ at j may result only if the node below produces t is the kind of semantic information – *how* the computation at j depends on the input from i – which might (should) be incorporated into the specification of dependencies. In short, if the dependency $s \rightarrowtail f(t)$ in C is not of the intended kind, it should be prohibited by the specification (this topic is to be explored more fully in a future paper).

Definition 4.11 For $\mathsf{A} \subseteq \mathsf{Alg}(\Sigma)$, $\mathsf{D} \subseteq \mathsf{DD}^n(\Sigma)$, an (A, D) n-algebra C is $\langle \bar{C}, \vec{C}, ev^C \rangle$ where

1. $\vec{C} \in Obj(\mathsf{D})$;
2. $\bar{C} \in Obj(\mathsf{A})$;
3. $ev^C : \vec{C} \times \mathcal{T}_\Sigma \to \bar{C}$ is a (partial) evaluation function such that $\forall t \in lab^{\vec{C}}(i) : ev^C(i, t) = t^{\bar{C}}$.

As for C-algebras, the function ev is uniquely determined by $\bar{C}$ and the labelling of $\vec{C}$.

Definition 4.12 An (A, D) n-homomorphism $h : C \to D$ is a pair $\langle \bar{h}, \langle \vec{h}, \vec{h}_l \rangle \rangle$ where

1. $\bar{h} : \bar{C} \to \bar{D} \in Mor(\mathsf{A})$;
2. $\langle \vec{h}, \vec{h}_l \rangle : \vec{D} \to \vec{C} \in Mor(\mathsf{D})$, and
3. $\bar{h}(ev^C(i)) = ev^D(\vec{h}_l(i))$, for all $i \in \vec{h}(I^{\vec{D}})$.

The last condition (illustrated in figure 19) is a generalisation of the commutativity condition for C-homomorphisms from the definition 2.8. In the deterministic case, the two coincide since then $\vec{h}_l$ is uniquely determined by $\vec{h}$.

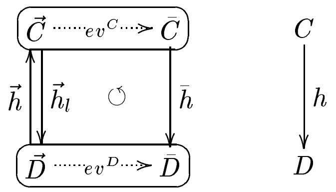


Figure 19: A homomorphism between two (A, D) n-algebras.

The category $\mathsf{CAlg}^n(\mathsf{A}, \mathsf{D})$ has (A, D) n-algebras as objects and (A, D) n-homomorphisms as morphisms. Implementations of $\mathsf{CAlg}^n(\mathsf{A}, \mathsf{D})$ are defined as before (definition 2.11), and so are the projection functors Dep^n and MAr^n. The lemma 2.15 generalizes trivially to the nondeterministic context by an entirely analogous proof using the construction (1). Modifying the first point of definition 4.5 to the requirement $\forall i \in I^{\vec{D}}\ \forall t \in lab^{\vec{C}}(\vec{h}(i)) : t^A = \vec{h}_l(t)^A$, we may define the class $\mathsf{CAlg}^n(\mathsf{A}, \mathsf{DD}^n(\mathsf{A}))$ of n-algebras with compatible dependency morphisms. The lemma 4.7 generalizes then equally easy to this class by a construction analogous to (8). We therefore do not do it explicitly here but only emphasize that the earlier results remain valid when we insert $\_^n$'s at whatever places we might wish.

4.5 Simulation, Implementation and Space-Time Representations

The simulation morphisms play a crucial role in defining appropriate ways of porting implementations. Although the category MAr allows arbitrary Gr^+-morphisms, in practice they will have to reflect more detailed properties of the actual architectures. Such properties may be directly expressible by the space-time representations but, in many cases, will appear as additional restrictions on the admissible morphisms – the space-time representation does not always capture all the relevant information concerning the possible computations on a given architecture.

4.5.1 Sending Multiple Data

Recall the morphism $m : W \to Z^+$ from figure 4.c) simulating a 2-dimensional vector W by a 3 processor ring Z. An alternative simulation n is illustrated in figure 20.

n maps distinct edges of W onto distinct but non-disjoint paths of Z. This will yield an adequate simulation provided that the latter allows us to join the data received from multiple sources and send them in one package at the time. The double arrows from the nodes $(Z_1, 2)$ and $(Z_1, 4)$ indicate that, at these points, data sent from different sources (e.g., from $(Z_1, 1)$ and $(Z_3, 1)$) have to be collected and sent together in one step. Of course, the same happens in figure 4.c) at the points $(Z_3, 3)$ and $(Z_3, 5)$. Whether this is possible, and how, will depend on the actual

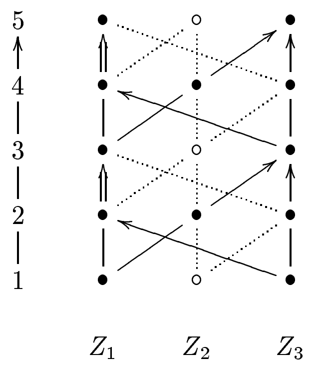


Figure 20: An alternative simulation $n : W \to Z^+$.

architecture. Consequently, the simulation morphisms have to take this phenomenon into account. In either case of this example this does not seem to cause any problems because the collected data are not communicated to another processor but kept at the one receiving them.

4.5.2 Broadcast vs. Stepwise Communication

A similar distinction will have to be made when a dependency is implemented on an architecture with broadcast vs. one where each processor can perform at most one communication at each time step (let's call the latter 1-ctp: one-communication-(at-)time-(per-)processor). For the sake of example, let's consider 3 processor architectures: broadcast allows arbitrary communications between arbitrary processors at each time step, while 1-ctp allows each processor to send data in at most one direction at each time step. (But at each step, any of the communications possible on a broadcast architecture is possible on the token architecture.) Suppose that the dependency D in question has points, i, such that there are several j_n with $i \rightarrowtail j_n$ for, let say, $n = 3$. It may be trivially implemented on a broadcast architecture U with 3 processors as illustrated in figure 21.b). A graphical space-time representation of a 3 processor 1-ctp V will be the same as that of U (21.a). However, for a given processor, V allows at most one communication at a time, and so the edges in figure 21.a) (except for the vertical ones) must now be interpreted with an "exclusive or" rather than with an "and". Consequently, implementation of D on V cannot look like in 21.b) but will require at least two time steps, as exemplified in 21.c).

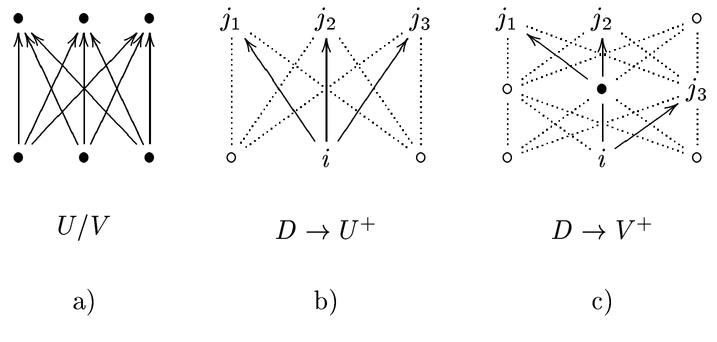


Figure 21: a) Space-time representation of a broadcast U and a 1-ctp V, b) implementation of D on U, and c) on V.

It is easier to think in terms of broadcast and so one may expect that implementations of dependencies will be made in terms of such architectures. Having a simulation $m : U \to V^+$, we will be able to port such an implementation to a 1-ctp architecture. The only point is that such an m must respect the fact that a space-time graph does not provide a fully faithful representation of the communication structure.[15]

4.5.3 Portability and Efficiency

Another issue concerns efficiency of the ported implementations. For instance, the implementation achieved in figure 6.c) as the result of porting an implementation from a vector to a ring, is certainly not the most efficient one. There is, for instance, the unnecessary step from $(Z_3, 1)$ to $(Z_3, 2)$ and, in general, from $(Z_3, 1+(4*n))$ to $(Z_3, 2+(4*n))$. Z_3 does not have to wait at the point $1+(4*n)$ to send data to Z_1, but can do it immediately. This delay occurs here because we have used a generic simulation m of W by Z. It is generic in the sense that it allows us to port *any* implementation from W to Z. In each particular case, like the present one, it may be possible to find a more efficient mapping which is tailored for this particular application. However, though efficiency of computation always is a concern, the need for portable and reusable code is an overriding urgency. What we have proposed is a sound methodology which guarantees *correctness preservation* of an implementation which is ported from one platform to another. We expect that the possible loss in computational efficiency will, hopefully, be richly compensated by increased *engineering efficiency*. And, having succesfully ported an implementation, there is nothing precluding some final optimization in order to achieve a more efficient product.

4.5.4 Code generation

We called a mapping from a data dependency to an architecture "implementation". This is a very generous name and we do not claim that what we have obtained is an actual implementation.

It is not at all obvious that the communication dependencies and the data dependencies should have the same or similar interpretation in terms of evaluation strategies. We may imagine that the hardware dependency $i \rightarrowtail j$ reflects the fact that j has to wait for a piece of data from i – once it is produced and received, j begins its computation. This view is more data dependency oriented – as exemplified by the Fibonacci function in figure 2. But we may also imagine a different scenario in which j ignores the input from i and simply evaluates whatever it is supposed to as soon as it has sufficient information to do so (i.e., a non-strict interpretation of the dependency). The latter is often the approach used in many parallel implementations.

A unifying view is to think of the nodes as "active agents" who make their local decisions as to how to treat the incoming information. Then the presence of an edge identifies a *potential* dependency. If a term is evaluated in a strict manner, where all the data being referenced have to be gathered before evaluation can proceed then the edge identifies the actual dependencies. But if the evaluation is non-strict, e.g., it is determined dynamically during actual computation whether a particular piece of data is required or not, then the computation will realise a subset of the data dependency edges. As a particularly interesting case, if we admit nondeterminism in computations, it may turn out that the dependencies are introduced exactly in order to allow non-strict evaluations where the relative speed of processing and communication determines which dependencies are realized. The resulting implementation may turn out to be more time-efficient, allowing simultaneous computation of possible inputs.

Although not sufficiently detailed, our notion of implementation (definition 2.11) does capture several aspects of parallelisation. Firstly, different threads in the space-time graph of a given architecture ($\{\langle s, t\rangle : t \in \mathbb{N}\}$) will represent different processors and so an implementation mapping amounts to *partitioning* of the code onto these processors.

Furthermore, the links between different components of the partition correspond to possible communications between processors. Mapping a particular edge of a data dependency onto a path in a space-time graph amounts then to a particular choice of *routing*. (This is why we want to work with the path, and not transitive closures of graphs.)

[15]The same problem will occur in many other cases. For instance, if we take the ring Z in figure 4.b) to be a token ring (only one processor "holding the token" can send data at a time passing, simultaneously, token to the receiving processor, which so can send data and the token to the next one). In spite of the same space-time representation, m would not then be a correct simulation of W since the full communication history on a token ring must form a path, which is not the case in 4.c).

Finally, implementation morphisms represent the generation of the actual code which consists of two parts: the communication part and the computation part. The shapes of dependencies serve as the source only of the first part which is related to the communication structure, while their labelling indicates distribution of the actual code. $i \rightarrowtail j$ could correspond to the following commands being executed at i and j, respectively: "i : `compute X;` **send to** `j;`", and "j : **wait for** `i; compute Y;`".[16] X and Y are the pieces of code resulting from the labelling of i and j, together with the semantics of these labels implied by the specification. Notice, however, that the separation is not absolute since the semantics of Y may, in addition, indicate that the successful completion of the "**wait for** `i;`" is either necessary or else merely optional for the execution of Y. In the former case the **wait** will be strict, in the latter, it may be time dependent.

Of course, our model comprises only the abstract notion, not such a concrete view of implementation. For instance, the actual code for each processor has to be generated by "sequentializing" the pieces of code assigned to the subsequent nodes on the path corresponding to the processor. Also, data dependencies contain explicit information about what is computed at each node, but this obviously does not apply to the communicating processors. The sending node i knows what the value of X it is sending, but X remains unknown to the recipent j until it has received it. Although one can imagine how this transition from a data dependency to the actual code might be accomplished, it certainly warrants a closer analysis. Code generation for a special class of C-algebras specified equationally with the explicit specification of the dependency structures was studied in connection with the SAPPHIRE methodology for development of parallel programs, [Hav93]. A more detailed investigation of both the specification formalism and parallel code generation is required to demonstrate the general applicability of the present theory.

5 Conclusions

We have introduced a general notion of communication algebras which extend the standard algebraic semantics with additional structures, data dependencies, carrying information concerning the evaluation strategy. The dependency morphisms can be used to design more efficient implementations, in particular, by syntactic memoisation. The combination of dependencies with actual algebras and the notion of C-homomorphisms allows us to perform semantic memoisation as well.

The explicit information about the computation structure opens the possibility for constructing parallel implementations from algebraic specifications. Two aspects of parallelisation: (a) the logical dependency decomposition and (b) partitioning and routing on a physical medium, have been expressed as, respectively, dependency and implementation morphisms. We have shown how indexing communication algebras with machine architectures yields a fibration. Its explicitly defined cleavage provides a way of porting implementations between various (parallel) architectures.

The general framework admits the definition of various subclasses of communication algebras. One such class – the algebras with compatible dependency morphisms, $\mathsf{CAlg}(\mathsf{A}, \mathsf{DD}(\mathsf{A}))$ – has been singled out and studied in more detail.

We have suggested a method allowing one to construct data dependencies from given specifications of recursive CR-functions. A special case of this method, combined with a subclass $(\mathsf{A}, \mathsf{D}^{\mathsf{Gr}}_{\Sigma})$ of graph-morphisms (cf. 4.3.2) and embedded in the programming language SAPPHIRE, has been worked out in more detail and used in practical applications for parallelisation of programs [Hav93, ČH95].

Future work

Most of the issues listed in section 4 need closer study. Summarizing, we can mention the following points:

[16] Whether the directions of sending/receiving are indicated by the processor identifiers or else only by the local communication channels is an even more specific information depending on the actual machine and programming language.

- Although we did not discuss any specification formalism, it should be obvious that all the aspects of our setting are amenable to a description in the usual algebraic way. In particular, data dependencies, architectures, and morphisms between them are standard algebraic objects.

- An important question concerns the ways of finding the data dependencies from a given algebraic specification. Section 3 opens a way in this direction. Some open questions concern the generality of the requirements listed at the beginning of 3.2 and, in particular, the syntactic conditions on the definitions like (5) which would ensure well-definedness of R in specific algebras. Construction of graphs and labellings from specifications is another issue involving finding the appropriate algebra A (which often, but not always, is an initial model of a specification).

  Alternatively, one might attempt using explicit specification of dependencies which could be collected in a library.

- We have indicated how the operational and multialgebraic notions of nondeterminism may be included into the framework. However, the role of nondeterminism will be explored further.

- The dependency and, especially, the simulation morphisms between existing architectures could be easily defined and stored for the future use in concrete applications. It would be desirable to develop such a library and the tools for its use.

- We would like to study closer the notion of dependency refinement and its possible use for the development of algorithms. It is possible that the techniques of graph transformation might be relevant for an alternative definition of dependency morphisms.

Acknowlegment

We thank Andrzej Tarlecki and Eric Wagner for interesting suggestions and comments on earlier versions of this paper. Kristoffer Rose deserves thanks for designing and making available the `XY-pic` package for drawing nice diagrams.

References

[BW90] M. Barr, C. Wells. *Category Theory for Computing Science.* Prentice Hall, (1990).

[Ben85] J. Bénabou. Fibred Categories and the Foundation of Naive Category Theory. *Journal of Symbolic Logic*, 50, (1985).

[BGT91] R. Burstall, J. Goguen, A. Tarlecki. Some Fundamental Algebraic Tools for the Semantics of Computation. Part 3: Indexed Categories. *TCS*, 91:239-264, (1991).

[CT65] J. W. Cooley, J. W. Tukey. An algorithm for the machine computation of complex Fourier series, *Math. Comp.* vol. 19, pp. 297-301, (1965).

[ČH95] V. Čyras, M. Haveraaen. Modular Programming of Recurrencies: a Comparison of Two Approaches, *Informatica*, vol. 6, no. 4, (1995).

[FOW87] J. Ferrante, K.J. Ottenstein, J.D. Warren. The Program Dependence Graph and its use in Optimization. *ACM ToPLaS*, vol. 9, no. 3, (1987).

[Hav90] M. Haveraaen. Distributing Programs on Different Parallel Architectures. *Proceedings of the International Conference on Parallel Processing, Software*, vol. II, (1990).

[Hav93] M. Haveraaen. *How to Create a Parallel Program without Knowing it.* Proceedings of the 4^{th} Nordic Workshop on Program Correctness, Tech. Rep. no. 78, University of Bergen, Department of Informatics, (1993).

[Her93] C.A. Hermida. *Fibrations, Logical Predicates and Indeterminates.* Technical Report DAIMI PB-462, Computer Science Department, Aarhus University, (1993).

[Hus93] H. Hussmann. *Nondeterminism in Algebraqic Specifications and Algebraic Programs.* Birkhäuser, (1993).

[Jac91] B. Jacobs. *Categorical Type Theory.* PhD thesis, University of Nijmegen, (1991).

[KK90] K. Kennedy, K. McKinley, Loop Distribution with Arbitrary Control Flow, *Supercomputing*, (1990).

[Mes90] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *TCS*, 96:73-155, (1992). (also J. Meseguer. *Rewriting as a Unified Model of Concurrency.* Technical Report SRI-CSL-90-02R, SRI International, 1990.)

[MW83] W.L. Miranker, A. Winkler. Spacetime Representations of Computational Structures. *Computing*, vol. 32, (1984).

[RB88] D. Rydeheard, R. Burstall. *Computational Category Theory.* Prentice Hall, (1988).

[WHM96] M. Walicki, M. Haveraaen, S. Meldal, *Communication Algebras*, Tech. Rep. no 117, Dept. of Informatics, University of Bergen, (1996).

[WM94] M. Walicki, S. Meldal. Multialgebras, Power Algebras and Complete Calculi of Identities and Inclusions. In *Recent Trends in Data Type Specification,* LNCS, vol. 906, (1994).

[WN93] G. Winskel, M. Nielsen, *Models for Concurrency*, TEMPUS Summer School, Brno, (1993).