Under consideration for publication in Math. Struct. in Comp. Science

Computation Algebras

Michał Walicki†
University of Bergen
Department of Informatics
P.Box 7800, 5020 Bergen, Norway
michal@ii.uib.no

Magne Haveraaen†
University of Bergen
Department of Informatics
P.Box 7800, 5020 Bergen, Norway
magne@ii.uib.no

Sigurd Meldal†
CalPoly
Computer Science Department
San Luis Obispo, CA 93407, USA
sigurd@calpoly.edu

Received 22 June 2000

We introduce a framework which generalizes algebraic specifications by equipping algebras with descriptions of *evaluation strategies*. The resulting abstract mathematical description allows one to model the *implementation* of algebras on various platforms in a way independent of the function-oriented specifications.
We are studying algebras with associated data dependencies. The latter provide separate means for modeling computational aspects apart from the functional specifications captured by an algebra. The formalization of evaluation strategies (1) introduces increased portability among different hardware platforms, and (2) allows 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 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 realized, need

† The authors gratefully acknowledge the financial support received from the Norwegian Research Council (NFR).

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 in 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 proposed method comes into successful play.

We propose a framework of *computation algebras* addressing the issue of how to incorporate computation-related aspects within the framework of classical algebraic semantics. Computation algebras allow the developer to transform abstract requirements (expressed in an algebraic specification) into designs by adding implementation-specific information concerning the order and dependency between various subcomputations. 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. (Miranker and Winkler 83; Ferrante *et al.* 87; Kennedy and McKinley 90). We proceed from the algebraic, that is, functional descriptions and dependencies relate functional terms and not program statements. In particular, a variable is only a (sub)term denoting a specific value – it is not an actual program variable of a program capable of storing various values at different points of computation.

We mainly address 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 among various parts of a computation enables us to determine its possible parallel distributions. In addition to algebras and data dependencies, we will introduce a repre-

sentation of (parallel) machine architectures and show how mappings from dependencies to architectures may be utilized to yield parallel implementations. The framework enables us then to *port the implementations of computation 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 the efficiency of the implementation.

Section 2 introduces the basic notions of computation algebras and their implementations and illustrates them on a simple example. It also gives a general view of the envisaged development process using computation algebras. Section 3 addresses the issue of constructing dependencies for a function defined by a set of recurrence equations over a given algebra. It illustrates the ideas of the programming language SAPPHIRE developed at the University of Bergen for which computation algebras provide the semantic framework.

Section 4 refines the picture of development sketched in section 2 – its results suggest a more specific metodology for development using computation algebras. Section 5 shows how nondeterminism can be included into the proposed framework. Section 6 summarizes the results and indicates some directions for future work.

The paper focuses on the description of the *concepts*; its technical content is limited to definitions and a few results. It gives an improved version of the results from (Walicki *et al.* 96). The more technical aspects assume some elementary knowledge of the category theory (e.g., (Mac Lane 71; Rydeheard and Burstall 88; Barr and Wells 90)) with particular emphasis on fibrations.

2. The Framework of Computation Algebras

This section introduces the framework of computation algebras. We define first some basic notions in Section 2.1, then data dependencies in Section 2.2, computation algebras in Section 2.3, and their implementation in Section 2.4. Section 2.5 summarizes these pieces presenting the view of the development process using computation 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 Section 4.1 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 multiagebras with various multihomomorphisms, power algebras, unified algebras, etc. We will define various notions relatively to an arbitrary full subcategory $\mathsf{A} \subseteq \mathsf{Alg}(\Sigma)$.[†]

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

The primary objects of our interest will be *simple directed acyclic graphs* which

— have at most one edge between any pair of nodes, and
— have no isolated nodes (each node has at least one incident edge).

We call them *simple DAGs*. We will, however, need a more general notion of a mapping of such graphs than a simple graph homomorphism. Mapping an edge of the source graph to a *path* in the target graph corresponds to splitting the computation into several steps. On the other hand, mapping an edge to a *set* of edges (or paths) corresponds to distributing the computation.

To define this concept of a morphism, we introduce three categories: DAG – of DAGs, DAG° – of path closures, and $\mathsf{DAG}^\oplus$ – of distributed paths (or just sets of paths). The latter two categories are used merely for the purpose of defining more general morphisms between DAG-objects. An object in DAG° contains, in addition to the underlying graph G, also all the paths over G – we will want to map an edge of one graph onto a path in a target graph.[‡] An object in $\mathsf{DAG}^\oplus$ contains, in addition, "parallel composition" of paths which, too, will be possible images of edges. The construction described in the rest of this subsection starts with the category DAG: the free functor F° adds freely to each object all the paths and then, the free functor F^+ adds all the sets of paths. The resulting composition $F^\oplus = F^\circ; F^+$ has left adjoint and our morphisms are Kleisli morphisms obtained from this adjunction.

Definition 2.1 DAG is the category of *directed acyclic graphs* where

1 objects are *DAGs* – algebras over signature $\Gamma = \langle I, E; s, t : E \to I \rangle$, where
   — I is a set of nodes and E a set of edges;
   — s/t return the source/target node of an edge;
   — E viewed as a relation ($\subseteq I \times I$) is acyclic;
   — there are no isolated nodes: for each node v, there is an edge e with $s(e) = v$ or $t(e) = v$.

2 morphisms are Γ-homomorphisms.

□

We will often write $i \rightarrowtail j$ to indicate the existence of an edge e from i to j (i.e., with $s(e) = i$ and $t(e) = j$). A Γ-algebra is a DAG (an object of DAG) if the transitive closure $\rightarrowtail^+$ of this relation is irreflexive.

A DAG may have several edges between a pair of nodes. Of particular importance to us will be *simple* DAGs – the ones with at most one edge between a pair of nodes.

Definition 2.2 **A** *simple DAG* is a *DAG* satisfying: $s(e_1) = s(e_2) \wedge t(e_1) = t(e_2) \Rightarrow e_1 = e_2$; □

Definition 2.3 DAG° is the category of *path closures* of DAGs where

[‡] The symbol '$\circ$' is used for (sequential) composition in diagrammatic order – not for functional composition.

1 objects are algebras over signature $\Gamma^\circ = \Gamma \cup \{_ \circ _ : E \times E \to E\}$ whose Γ-reducts belong to DAG – for any $P \in Obj(\mathsf{DAG}^\circ) : P|_\Gamma \in Obj(\mathsf{DAG})$. $_ \circ _$ is a partial operation of path formation defined for any edges e_1, e_2 with $t(e_1) = s(e_2)$, and satisfying

— $e_1 \circ (e_2 \circ e_3) = (e_1 \circ e_2) \circ e_3$
— $s(e_1 \circ e_2) = s(e_1)$ and $t(e_1 \circ e_2) = t(e_2)$;

2 morphisms are Γ°-homomorphisms.

□

Thus, a DAG°-morphism $f : G \to H$ maps an edge $x \rightarrowtail y$ from G onto en edge from $f(x)$ to $f(y)$ in H – this latter edge, however, may correspond to a path obtained by composition of several edges in H. The following fact is easy to verify:

Proposition 2.4. Let

— $F^\circ : \mathsf{DAG} \to \mathsf{DAG}^\circ$ be the free functor sending a $G \in Obj(\mathsf{DAG})$ onto its *path closure* G° by adding freely all possible compositions of edges from G; for a DAG-morphisms h, we let $F^\circ(h) = h$ [§]
— $U^\circ : \mathsf{DAG}^\circ \to \mathsf{DAG}$ be the forgetful functor sending a graph G° onto its Γ-reduct $G^\circ|_\Gamma$; for a DAG°-morphisms $h : U^\circ(h) = h$.

There is an adjunction $F^\circ \dashv U^\circ$ with the unit being the inclusion of the source graph into its path closure.

The objects in the next category will allow two nodes to be connected not only by paths but also by all finite, non-empty sets of paths.

Definition 2.5 $\mathsf{DAG}^\oplus$ is the category of *distributed* path closures of DAGs where

1 objects are algebras over signature $\Gamma^\oplus = \Gamma^\circ \cup \{_ \oplus _ : E \times E \to E\}$ whose Γ°-reducts belong to DAG° – for any $P \in Obj(\mathsf{DAG}^\oplus) : P|_{\Gamma^\circ} \in Obj(\mathsf{DAG}^\circ)$. $_ \oplus _$ is a partial operation of parallel composition defined for any edges e_1, e_2 with $s(e_1) = s(e_2)$ and $t(e_1) = t(e_2)$, which is associative, commutative and idempotent

— $e_1 \oplus (e_2 \oplus e_3) = (e_1 \oplus e_2) \oplus e_3$
— $e_1 \oplus e_2 = e_2 \oplus e_1$
— $e \oplus e = e$

composition distributes over $\oplus$:

— $e_1 \circ (e_2 \oplus e_3) = (e_1 \circ e_2) \oplus (e_1 \circ e_3)$
— $(e_1 \oplus e_2) \circ e_3 = (e_1 \circ e_3) \oplus (e_2 \circ e_3)$;

and since $\oplus$ is defined only for edges with the same source and target:

— $s(e_1 \oplus e_2) = s(e_1) = s(e_2)$ and $t(e_1 \oplus e_2) = t(e_1) = t(e_2)$;

2 morphisms are $\Gamma^\oplus$-homomorphisms

□

[§] Strictly speaking, $F^\circ(h)$ is h extended to paths: $F^\circ(h)(e_1 \circ e_2) = h(e_1) \circ h(e_2)$.

The first three axioms allow formation of sets of edges (which here may be paths from the "underlying" DAG); the next two ensure that composition of sets of paths corresponds to forming sets of composite paths. Canonically, an edge $x \rightarrowtail y$ is a non-empty set of edges (paths in the "underlying" DAG) from x to y. Hence, a morphism $f : G \to H$ maps en edge $x \rightarrowtail y$ from G onto a non-empty set of paths between $f(x)$ and $f(y)$ in H. Again, we have an easy fact:

Proposition 2.6. Let

— $F^+ : \mathsf{DAG}^\circ \to \mathsf{DAG}^\oplus$ be the free functor sending a $G^\circ \in Obj(\mathsf{DAG}^\circ)$ onto $G^\oplus \in Obj(\mathsf{DAG}^\oplus)$ by adding freely all possible parallel ($\oplus$) compositions of edges from G° subject to the restrictions and axioms from the above Definition;

— $U^+ : \mathsf{DAG}^\oplus \to \mathsf{DAG}^\circ$ be the forgetful functor sending a graph $G^\oplus$ onto its Γ°-reduct $G^\oplus|_{\Gamma^\circ}$;

— both functors map morphisms onto identical morphisms in their respective target categories.

There is an adjunction $F^+ \dashv U^+$ with the unit being the inclusion of the source graph into its closure under parallel composition.

Now define two functors

$$F^\oplus = F^\circ; F^+ : \mathsf{DAG} \to \mathsf{DAG}^\oplus \quad \text{and} \quad U^\oplus = U^+; U^\circ : \mathsf{DAG}^\oplus \to \mathsf{DAG}. \tag{1}$$

The two adjunctions from Propositions 2.4 and 2.6 give an adjunction $F^\oplus \dashv U^\oplus$. The morphisms we are looking for can be now defined as Kleisli morphisms between objects of DAG induced by this last adjunction. The resulting category is given by: the objects are simple DAGs, a morphism $f : G \to H$ is a DAG-morphisms $f : G \to U^\oplus(F^\oplus(H))$. Morphisms are composed pointwise: for $f : A \to B$ and $g : B \to C$, f maps an edge $e = x \overset{A}{\rightarrowtail} y$ onto a set of paths $f(e) = \{p_1, p_2 ... p_n\}$ from $f(x)$ to $f(y)$ in B. Then, g maps each path p_i onto a set of paths $g(p_i) = \{r_{i1}, r_{i2} ... r_{ik}\}$ from $g(f(x))$ to $g(f(y))$ in C, obtained by mapping each edge of p_i onto a set of paths and composing these paths along the p_i. The composition is then $g(f(e)) = \bigcup_{p_i \in f(e)} g(p_i)$. Formally

Definition 2.7 The category Gr is the full subcategory – with simple DAGs as objects – of the Kleisli category induced by the adjunction $F^\oplus \dashv U^\oplus$. ¶ □

Since we will be working with the category Gr, in the sequel, we will use the notation $G^\oplus$ – where $G \in Obj(\mathsf{Gr})$ – as an abbreviation for $U^\oplus(F^\oplus(G))$.

Example 2.8 Three examples of Gr-morphisms are given in Figures 1 and 2. The former maps edges onto paths while the latter maps an edge onto distributed paths. □

A Gr-morphism $h : A \to B$ expresses the fact that B can *simulate* A – edges of A are represented in B not necessarily as edges, but possibly as paths, or even *sets* of paths. B may also contain additional edges not in the image of h.

¶ This concise formulation expresses what we have described above. For the construction of Kleisli category see e.g. (Mac Lane 71).

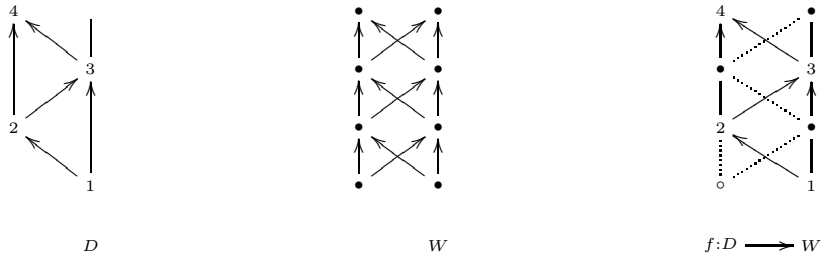


Fig. 1. A Gr-morphism from D to W.

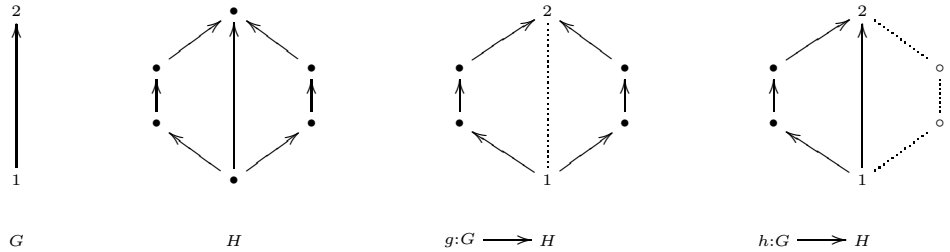
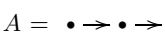
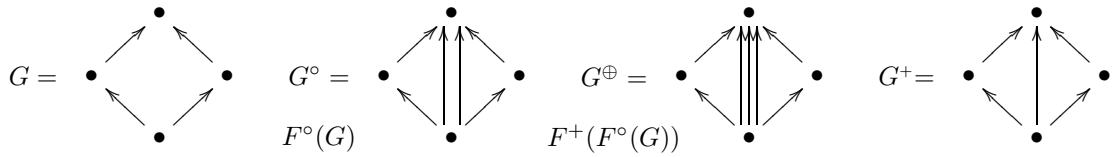


Fig. 2. Two Gr-morphisms from G to H.

Remark. DAG-embedding of A into B is a special case of a Gr-morphism $A \to B$. One might expect that a surjective (on the nodes) DAG-morphism $B \to A$ would yield a simpler notion. 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 DAG-morphism from $B =$ onto $A =$  . But we would like to say that B does reflect the edges of A (and introduces some more) – and so there is a Gr-morphism $A \to B$. □

From now on, we will treat "graph" as a synonym for "DAG" since DAGs are the only graphs we are going to deal with. Also, our graphs are primarily connected *simple* DAGs, in which case we will often identify a graph G and its edge relation E^G, written also $\overset{G}{\rightarrowtail}$. Then, we say that G is *well-founded* when the relation $\overset{G}{\rightarrowtail}$ is, and denote by G^+ the transitive closure $\overset{G^+}{\rightarrowtail}$. Notice that, in general, $G^\circ \neq G^+ \neq G^\oplus$. The following example (suggested by an anonymous referee) illustrates the difference between the three:



Occasionally, we may use the following terms:

— A graph is *sequential* if $\rightarrowtail^+$ 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-graph* ("space-time") is a simple DAG with nodes indexed by pairs $\langle s, t\rangle \in S \times \mathbb{N}$, 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*-graphs are singled out because they will be used as the **s**pace-**t**ime representation of hardware architectures (Miranker and Winkler 83). 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 (for some fixed $k > 0$).

We note the following characterization:

Proposition 2.9. Let $k : A \to B \in Mor(\mathsf{Gr})$. k is mono $\Leftrightarrow$ k is injective on the edges.

Proof. Let T be an arbitrary object and $f, g : T \to A$ two arbitrary morphisms.

$\Rightarrow$) If k is mono, then for any $f, g : T \to A$, $f; k = g; k \Rightarrow f = g$. In particular, for any edge $e \in E^T : k(f(e)) = k(g(e)) \Rightarrow f(e) = g(e)$. Taking T as a one-edge graph and considering all $f, g : T \to A$, implies that k must be injective on the edges.

$\Leftarrow$) Suppose k is injective on the edges and $f; k = g; k$. For each edge $e \in E^T$ we then have $k(f(e)) = k(g(e)) \Rightarrow f(e) = g(e)$. To verify that $f = g$, we have to check that also for each node i in T, $f(i) = g(i)$. Assume that for some node $f(i) \neq g(i)$. Since T is a DAG (according to Definition 2.1 it has no isolated nodes), there is an edge e incident to i and then we would have $f(e) \neq g(e)$. Because of the injectivity on the edges, we then get $k(f(e)) \neq k(g(e))$, contrary to the assumption $f; k = g; k$.

□

2.2. *Data Dependencies*

As remarked in introduction, our data dependencies are intended to capture the information concerning what resources are going to be used in performing new steps of computation. Resources in this context refer exclusively to the values of some functions and computations compute such values. Thus our data dependencies relate functional terms and not program statements. For instance, computation of the value of the term $f(s(x), 2)$ may depend on the value of $s(x)$ which, in turn, may depend on x. x, however is just a term – a mathematical variable which, in any instance, has some specific value. It 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 (simple DAG) with nodes labeled

by ground Σ-terms $\mathcal{T}_\Sigma$. The only requirement we put on the morphisms between such graphs is that they respect the sorting of labels.

Definition 2.10 **[Data dependencies]** $\mathsf{DD}(\Sigma)$ is the category of Σ *data dependencies* where:

1 objects are Σ data dependencies – pairs $\langle G, lab\rangle$ where $G \in Obj(\mathsf{Gr})$ is a simple DAG and $lab : I^G \to \mathcal{T}_\Sigma$ is a function labeling nodes of this graph with ground Σ-terms;
2 a $\mathsf{DD}(\Sigma)$-morphism $m : \mathbf{C} = \langle C, lab_\mathbf{C}\rangle \to \mathbf{D} = \langle D, lab_\mathbf{D}\rangle$ is a Gr-morphism between the underlying graphs $m : C \to D$, such that $\forall i \in I^C : Sort(lab_\mathbf{D}(m(i))) = Sort(lab_\mathbf{C}(i))$.[‖]

□

In the sequel, D will denote an arbitrary subcategory of $\mathsf{DD}(\Sigma)$. The functor

$$Gr : \mathsf{DD}(\Sigma) \to \mathsf{Gr} \tag{2}$$

is the obvious forgetful functor which, for a given Σ data dependency $\mathbf{D} = \langle D, lab\rangle$, returns its underlying graph D. We call such a graph the *shape of* $\mathbf{D}$.

The relation $i \rightarrowtail j$ (in the shape $Gr(\mathbf{D})$ of $\mathbf{D} = \langle D, lab\rangle$) indicates 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.

Example 2.11 To give a flavor of our aims, consider a specification of the Fibonacci function:

$$\begin{array}{lll}
\textsc{Fib}\ \textbf{is}\ \mathcal{S}: & Nat & \\
\mathcal{F}: & \ldots\text{the usual functions for } Nat & \\
& F: \quad Nat \quad \to \quad Nat & \\
\mathcal{A}: & F(0) \quad = \quad 1 & \\
& F(1) \quad = \quad 1 & \\
& F(n+2) \quad = \quad F(n) + F(n+1) &
\end{array}$$

Ignoring for the moment details (to be discussed later), we may easily recognize the dependency given by the tree of recursive calls $\mathbf{R}$ in Figure 3. Certainly, Definition 2.10 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 uninteresting morphisms admitted, in principle, by Definition 2.10. In Figure 3 we have indicated two dependency 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.[††]

Here, we may consider the dependency $\mathbf{D}$ as a refinement (improvement) of $\mathbf{R}$. Notice that $\mathbf{D}$ is sequential under the ordering $F(n) < F(m)$ for $n < m$. Nevertheless, the partitioning of $\mathbf{D}$ indicated by the vertical arrows implies serious savings of storage

[‖] 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.

[††] In Section 4.2.1, we suggest a more specific notion of a dependency morphism *compatible* with a given class of algebras of which d'' is an example.

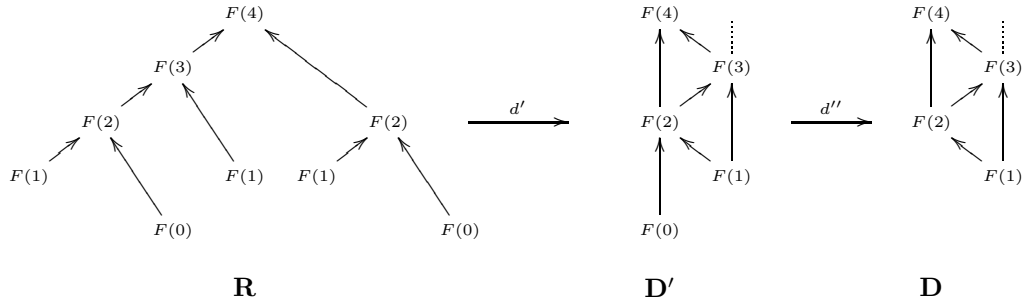


Fig. 3. The recursive tree dependency $\mathbf{R}$, together with the "actual" dependencies for computation of Fibonacci, $\mathbf{D}'$ and $\mathbf{D}$, and the dependency morphism $d = d'; d''$.

space, even if the computations along both lines are heavily synchronized and far from independent. □

Remark. As a matter of fact, there is also a dependency morphisms $g : \mathbf{D} \to \mathbf{R}$ which embeds $\mathbf{D}$ into the leftmost branch of $\mathbf{R}^{\oplus}$. 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. *Computation Algebras*

A Σ-computation 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)-computation algebra is an A-algebra with an associated D-dependency.

Definition 2.12 [Computation algebra] An (A, D)-*computation algebra* C is a triple $\langle A, \mathbf{B}, ev\rangle$ where

1 $A \in Obj(\mathsf{A})$;
2 $\mathbf{B} \in Obj(\mathsf{D})$;
3 $ev : I^{Gr(\mathbf{B})} \to A$ is an evaluation function such that $lab_{\mathbf{B}}(i) = t \Rightarrow ev(i) = t^A$.

□

The ev function is uniquely determined by the algebra A and the labeling of $\mathbf{B}$, so we will usually write a computation algebra as a pair $\langle A, \mathbf{B}\rangle$. When (A, D) does not matter or is clear from the context, we will simply write *computation algebra* (instead of (A, D)-computation 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 $\mathbf{B}$ of a computation algebra $\langle A, \mathbf{B}\rangle$ as its *communication* part.

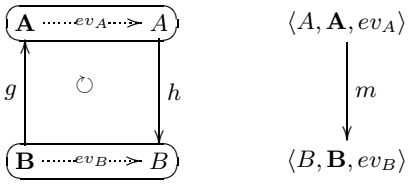


Fig. 4. A homomorphism between two computation algebras.

Definition 2.13 [Computation homomorphisms] An (A, D)-computation *homomorphism* $m : \langle A, \mathbf{A}, ev_A \rangle \to \langle B, \mathbf{B}, ev_B \rangle$ is a pair $\langle h, g \rangle$ such that (cf. Figure 4):

1 $h : A \to B \ \in Mor(\mathsf{A})$;
2 $g : \mathbf{B} \to \mathbf{A} \ \in Mor(\mathsf{D})$, and
3 $h(ev_A(g(i))) = ev_B(i)$, for all $i \in I^{Gr(\mathbf{B})}$.‡‡

□

As with computation algebras, when (A, D) does not matter, or is clear from the context, we will speak about *computation homomorphisms*.

The primary intuition behind this definition of homomorphism is that it is to be used to model a refinement process – in the above definition, we would say that $\langle A, \mathbf{A} \rangle$ refines $\langle B, \mathbf{B} \rangle$. Refinement concerns both the algebra and dependency part. The usual algebra homomorphism $h : A \to B$ corresponds to the classical notion of data refinement – A may have multiple representations for a single value from B. The (contravariant) homomorphism $g : \mathbf{B} \to \mathbf{A}$ represents the possible improvement of the dependency $\mathbf{B}$ by the dependency $\mathbf{A}$ (cf. Figure 3) – $\mathbf{A}$ may introduce paths, even sets of paths, in the place of single edges in $\mathbf{B}$.

In the example 2.11, the standard algebra of natural numbers $\mathcal{N}$ with function F, may give rise to three different computation algebras $\langle \mathcal{N}, \mathbf{R} \rangle$, $\langle \mathcal{N}, \mathbf{D}' \rangle$ and $\langle \mathcal{N}, \mathbf{D} \rangle$. The respective morphisms obtained from Figure 3, e.g. $\langle id_{\mathcal{N}}, d \rangle : \langle \mathcal{N}, \mathbf{D} \rangle \to \langle \mathcal{N}, \mathbf{R} \rangle$, will represent gradual refinements of $\langle \mathcal{N}, \mathbf{R} \rangle$. 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 computation homomorphism unless algebra $\mathcal{N}$ satisfies some additional (and here, unintended) equations.

Thus, refinement corresponds to a standard (data) refinement on the algebra part, while dependencies may be refined by "stretching" (to paths), "distributing" (to sets of paths) or augmenting with new dependencies. The following two examples illustrate these aspects of refinement.

Example 2.14 We let $\vec{C}$ and $\vec{D}$ be as in Figure 5, with $\vec{h}(a) = a$, $\vec{h}(b) = d$, 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

‡‡ The diagram in Figure 4 commutes, but not in a category (at least none we have defined explicitly), and so we write the composition of functions explicitly as $a(b(i))$.

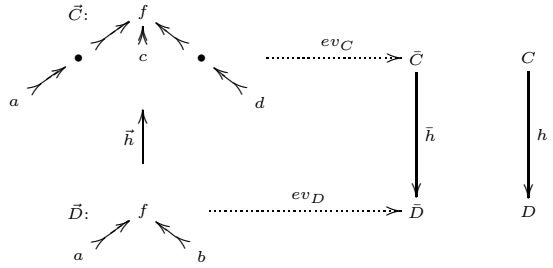


Fig. 5. Refinement of a computation algebra by adding new dependencies.

$\vec{h}(b) = d$ (taking, for simplicity, the names of the nodes in the Figure to be their labels), only if d evaluates in $\bar{C}$ to a value $ev_C(d) = x$ representing $ev_D(b)$, i.e., such that $\bar{h}(x) = ev_D(b)$.

Also, refinement "does not care" about what is computed in C at the nodes which are not in the image of $\vec{h}(\vec{Gr}(D))$. There may be new nodes and dependencies, also on nodes labeled by terms not occurring in the labeling of $\vec{D}$ (like $c \rightarrowtail f$ in $\vec{C}$). In particular, if we admit a computation algebra with an empty dependency part (i.e., empty carriers in the category D), a standard algebra $\bar{A}$ may be considered a computation 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 $\bar{h} : \bar{B} \to \bar{A}$. □

Example 2.15 As an example of possible "distribution", consider an algebra D with the sorts S, S_1, S_2, and the dependency $t \rightarrowtail f(t)$ for two terms of sort S (Figure 6). 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 ordinary equalities for these functions and, in addition, let $\bar{D} \models t = \langle t_1, t_2 \rangle$, and $\bar{D} \models f(t) = \langle f_1(t_1), f_2(t_2) \rangle$. This allows us to design the refinement of the dependency structure illustrated in Figure 6 (cf. Figure 2) which together with the identity on $\bar{D}$ yields a computation homomorphism $\langle id_{\bar{D}}, \vec{h} \rangle$. The morphism $\vec{h}$ refines the dependency $t \rightarrowtail f(t)$ in $\vec{D}$ to *both* paths (through t_1 and t_2). □

Computation algebras and their homomorphisms form a category.

Definition 2.16 [Category of computation algebras] $\mathsf{CAlg}(\mathsf{A}, \mathsf{D})$ is the category with objects being (A, D)-algebras and morphisms being (A, D)-homomorphisms (and obvious pointwise composition). When talking about the entire category $\mathsf{CAlg}(\mathsf{Alg}(\Sigma), \mathsf{DD}(\Sigma))$, we will instead write $\mathsf{CAlg}(\Sigma)$. □

We define two obvious functors:

$$\vec{_} : \mathsf{CAlg}(\mathsf{A}, \mathsf{D})^{op} \to \mathsf{D} \tag{3}$$

projects computation algebras onto their communication part, i.e., $\langle A, \vec{\mathbf{A}} \rangle = \mathbf{A}$ and

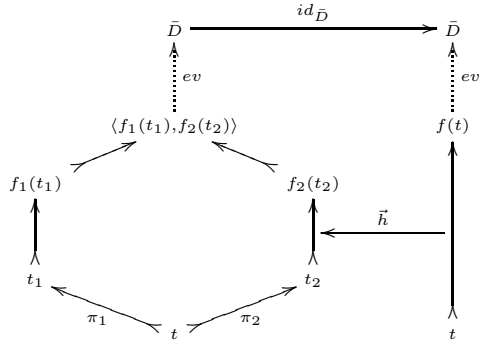


Fig. 6. Refinement of a computation algebras by distributing dependencies.

$\langle \bar{h}, \vec{g}\rangle = g$ with the notation from Definition 2.13.[§§] Similarly

$$\bar{_} : \mathsf{CAlg}(\mathsf{A}, \mathsf{D}) \to \mathsf{A} \tag{4}$$

projects computation algebras onto their algebra part ($\langle \bar{A}, \mathbf{A}\rangle = A$ and $\langle \bar{h}, g\rangle = h$).

We will now use these functors to simplify the notation for the components of a computation algebra: A representing such an algebra will denote the triple $\langle \bar{A}, \vec{A}, ev_A\rangle$, and a computation homomorphism h a pair $\langle \bar{h}, \vec{h}\rangle$.

2.4. *Machines and Distribution*

We introduce the model of machine architectures and their mappings (simulations) in Section 2.4.1, and then define mappings of a computation algebra to a given architecture called "distributions", Section 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 (Miranker and Winkler 83). Thus, an architecture is simply an *st*-graph, 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 graphs rather than the *st*-graphs) but here 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 set of chains of communications in Z. This is captured by the existence of a mapping of W into the (possibly distributed) path closure of Z, that is, a Gr-morphism $W \to Z$. Consequently, we define:

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

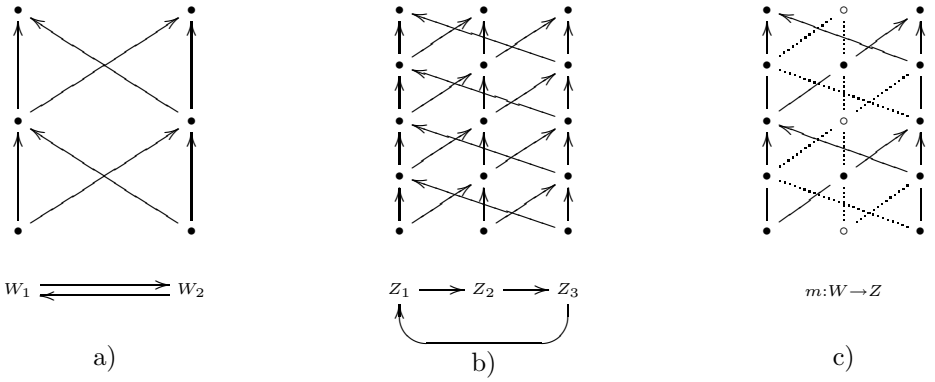


Fig. 7. Space-time representations of a) 2-dimensional vector W, b) 3-processor ring Z, and c) a simulation m of W by Z.

Definition 2.17 [**Machine architectures**] The category of *machine architectures* is Gr. □

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

We might have given a more restrictive definition of machine architectures. For instance, the unfoldings of actual processor-architectures will be *st*-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 distributions as morphisms of the same category Gr.

Example 2.18 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 neighbor: $W_1 \rightleftarrows W_2 \rightleftarrows \cdots \rightleftarrows W_n$. A space-time unfolding for a 2-dimensional vector is illustrated in Figure 7.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 7.b). This architecture may simulate the 2-dimensional vector W by the morphism m indicated in Figure 7.c). □

2.4.2. *Distribution.* Assume that we would like to implement a computation algebra on some specific machine architecture. *Distribution* is what, in our abstract setting, represents an implementation. It certainly does not capture all aspects of an actual implementation, and for this reason we have chosen to call it "distribution". This term reflects the intuition of distributing the dependencies of an algebra onto the communication structure of a given architecture.

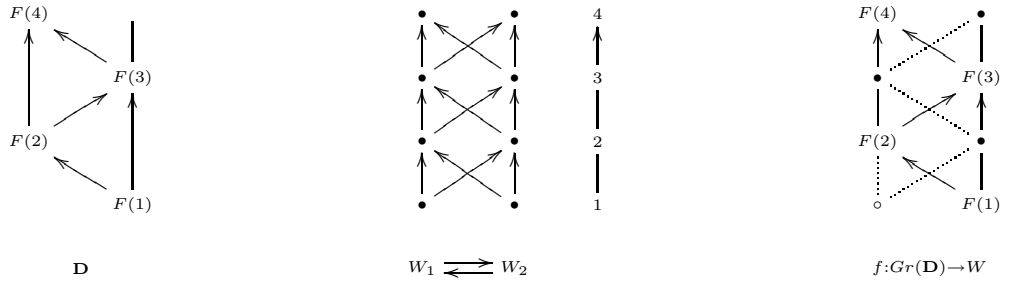


Fig. 8. a) Data dependency **D**, b) space-time representation of a 2-dimensional vector W, and c) a distribution f of **D** on W.

The composition of functors (2) and (3) $\vec{-}; Gr : \mathsf{CAlg}(\mathsf{A}, \mathsf{D})^{op} \to \mathsf{D} \to \mathsf{Gr}$ applied to an algebra returns the underlying graph of its communication part. We will write this composition as $\vec{Gr}$, i.e., $\vec{Gr}(X) = Gr(\vec{X})$, for an object or morphism X in $\mathsf{CAlg}(\Sigma)$.

Distribution of an algebra $\langle \bar{C}, \vec{C} \rangle$ on a given architecture W amounts to a simulation of the (shape of the) dependency structure given by $Gr(\vec{C})$ on the communication structure W.

Definition 2.19 [Distribution] A *distribution* $m_{C \to W}$ of a computation algebra C on an architecture $W$is a Gr-morphism $m : \vec{Gr}(C) \to W$. □

Example 2.20 The dependency **D** (with the shape $Gr(\mathbf{D})$ from Figure 3) is mapped onto a 2-processors vector W as it was illustrated in example 2.8, Figure 1. Figure 8 is just a repetition of this earlier example: a) is the data dependency **D** with the shape $Gr(\mathbf{D})$, b) is a 2-processors vector W and c) is a distribution f of **D** on W. □

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

Example 2.21 The mapping f from Figure 8.c) is a distribution of a computation algebra $D = \langle N, \mathbf{D} \rangle$ (with N a Σ_{FIB}-algebra satisfying FIB) on the vector W. Then, the mapping m (Figure 7.c) gives a ported distribution of D on the ring Z. The results are shown in Figure 9. □

A morphism between two distributions is defined as follows:

Definition 2.22 [Morphisms of distributions] A morphism between two (A, D) distributions, $m : f_{A \to W} \to g_{B \to Z}$, is a pair $\langle h, h' \rangle$, where:

1 $h : A \to B \in Mor(\mathsf{CAlg}(\mathsf{A}, \mathsf{D}))$;
2 $h' : Z \to W \in Mor(\mathsf{Gr})$, and
3 $\vec{Gr}(h); f = g; h'$.

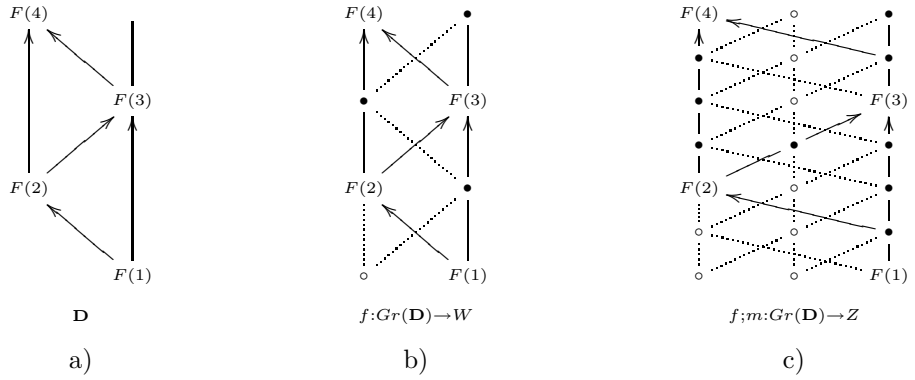


Fig. 9. a) Data dependency **D**, b) the distribution f of **D** on a 2-dimensional vector W, and c) distribution f ported to the ring Z.

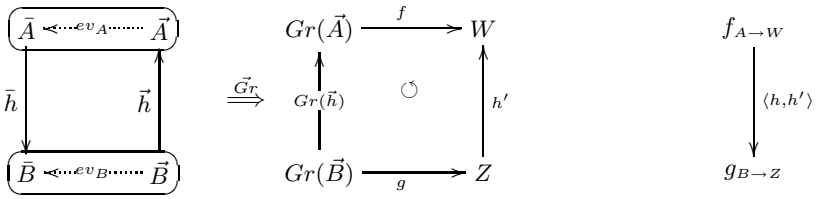


Fig. 10. A morphism $\langle h, h' \rangle$ of distributions.

□

The situation is depicted in Figure 10. The third condition is the requirement of the commutativity of the rightmost square in Gr.[¶¶]

Definition 2.23 We let $\mathsf{Dist}(\mathsf{A}, \mathsf{D})$ be the category

1 objects – distributions of (A, D) computation algebras (Definition 2.19),
2 morphisms given in Definition 2.22 and pointwise composition.[‖‖]

□

The forgetful functor

$$Arch : \mathsf{Dist}(\mathsf{A}, \mathsf{D}) \to \mathsf{Gr}^{op} \tag{5}$$

is the obvious projection of distributions onto their architecture part, i.e. $Arch(f_{A \to W}) =$

[¶¶] We let the distribution morphisms go in the opposite direction as the respective Gr-morphisms between the distributions. This choice is to keep with the convention that morphisms between structures reflect some form of "refinement" of the target by the source.

[‖‖] This works because the commutativity of the square in Figure 10 – i.e., for each edge e in $\vec{Gr}(B)$, the (sets of) paths obtained in W by $h'(g(e))$ and by $f(\vec{Gr}(h)(e))$ are identical – implies commutativity for each (set of) paths in $\vec{Gr}(B)$. Hence composition with another such commuting square, given by $\vec{Gr}(n) : \vec{Gr}(C) \to \vec{Gr}(B)$, $n' : Y \to Z$, and $k : Gr(\vec{C}) \to Y$, will yield a commuting square $(\vec{Gr}(n); \vec{Gr}(h)); f = k; (n'; h')$.

W and $Arch(\langle h, h' \rangle) = h'$. (Again, we may view each $Arch_{\mathsf{Dist(A,D)}}$ as a restriction of $Arch_{\mathsf{Dist(CAlg}(\Sigma))}$ to $\mathsf{Dist(CAlg(A, D))}$.)

2.5. *The Development Process*

The introduced framework is motivated by the view of the development of (possibly parallel) implementations by means of computation algebras. We envision the process starting with the usual refinement of algebraic specifications which proceeds through a series of subclasses of $\mathsf{Alg}(\Sigma)$ and results in one class of isomorphic algebras (an algebra $\bar{A}$ in Figure 11 is a chosen representative from this resulting class). At this point, when the specification has become sufficiently concrete, one introduces the dependencies, i.e., constructs a computation algebra $\langle \bar{B}, \vec{B} \rangle$. Dependencies can then be refined along with a further refinement of the algebra itself or, perhaps, keeping the algebra fixed.

Remark. 2.24. Since there is no surjectivity requirement on ev, the communication part $\vec{A}$ need not contain the full information about all operations of the algebra $\bar{A}$. (In the extreme case, the shape of $\vec{A}$ 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 $\bar{A}$ as a computation algebra with 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. Furthermore, we may imagine a specification of computation algebras as a kind of algebraic specification with hidden functions: only the visible operations are of interest to the implementation, and the resulting computation algebra need only provide the dependency structure for these operations. Finally, we can use this setting in situations where (parts of) an algebra are implemented on some machine (e.g. all built-in types constitute a given algebra), and one need to compute some additional, newly defined functions – the communication part will then be needed only for these new functions. □

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 distribute the dependencies $\vec{C}$, $f_{C \to W}$. Once this is done, there remains only the question of portability, that is, of moving the distribution to other 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 distributing 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 11 illustrates the process in its full generality, where *all* the aspects may change at *all* levels.

Although the technical results below are rather straightforward 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 computation algebras by machine architectures. More importantly, they give us the possibility of modularizing the implementation process and reusing its results.

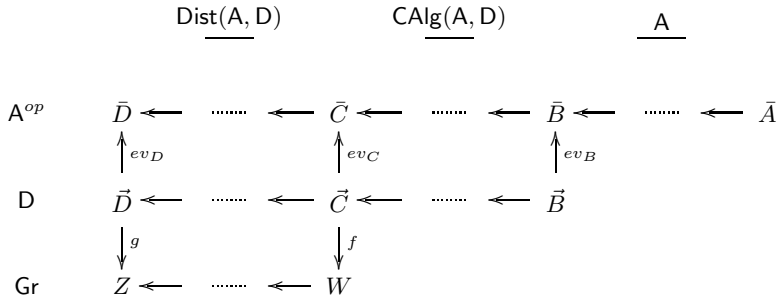


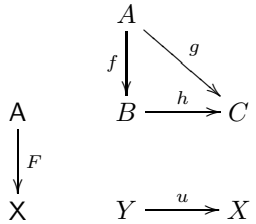
Fig. 11. The development process.

The simulation mappings between architectures (the Gr-morphisms) may be defined and stored independently of the actual programs implemented on them. Suppose that we have distributed a computation algebra D on an architecture W, $d_{D\to W}$ (Definition 2.19). In order to distribute D on another architecture Z, we need to define an appropriate distribution $c_{D\to Z}$. However, if we have a Gr-morphism $h : W \to Z$, it will tell us how Z can simulate any computation on W. Thus it will allow us to port the distribution $d_{D\to W}$ by just using this general simulation h.

Such a reuse of (stored) simulation morphisms between architectures for porting distributions will be formalized by showing that the functor *Arch* from (5) yields a *fibration*. Intuitively, this means that we can view the category Dist(A, D) as classes of computation algebras indexed by machine architectures – the *fiber* over a given architecture W represents all computation algebras which can be distributed on W. Then, given an architecture morphism $h : W \to Z$ and an algebra D distributed on Z (in Z's fiber), there is a canonical way of defining a *cleavage*: an algebra C and its distribution $c_{C\to Z}$ on Z together with a distribution morphism $cl(h, D) : c_{C\to Z} \to d_{D\to W}$ which is "compatible" with the simulation h. As a matter of fact, the triviality of the proof (that *Arch* is a fibration) coincides with the desired fact that $D = C$. Thus, cleavage gives explicitly the distribution resulting from porting a distribution of D from W to Z "along" a given simulation h.

The concept of fibration seems a very natural model for porting distributions and we will encounter it again later on. We now recall the notion of fibration, Definition 2.25, (for more on fibrations, see (Bénabou 85; Jacobs 91; Hermida 93)) and show that *Arch* is a fibration in Proposition 2.26.

Definition 2.25 **[Fibration]** Given two categories A and X and a functor $F : \mathsf{A} \to \mathsf{X}$:



1 A morphism $h : B \to C$ in A is *cartesian* if for any morphism $g : A \to C$ with $F(g) = F(h)$, there exists a unique $f : A \to B$ such that $f; h = g$.

2 A functor F is a *fibration* if for every $C \in Obj(\mathsf{A})$ and $u : Y \to F(C) \in Mor(\mathsf{X})$, there exists a cartesian morphism $h : B \to C$, and the composite of two cartesian morphisms is cartesian.

3 F is a *cofibration* if $F^{op} : \mathsf{A}^{op} \to \mathsf{X}^{op}$ is a fibration; it is *bifibration* if it is both fibration and cofibration.

4 An $A \in Obj(\mathsf{A})$ is *over* Y if $F(A) = Y$; a morphism $g \in Mor(\mathsf{A})$ is *over* u if $F(g) = u$. For an object $Y \in Obj(\mathsf{X})$, *the fiber* over Y, $F^{-1}(Y)$, is the subcategory of A with objects all those over Y and morphisms all those over id_Y.

5 A particular choice of a cartesian lifting for every appropriate morphism $u \in Mor(\mathsf{X})$ is called a *cleavage* for F. Given a cleavage cl, a morphism $u : Y \to X \in Mor(\mathsf{X})$ and an object $C \in Obj(\mathsf{A})$ over X, we denote by $cl(u, C)$ the cartesian lifting of u whose codomain is C.

We will mention *split* fibrations:

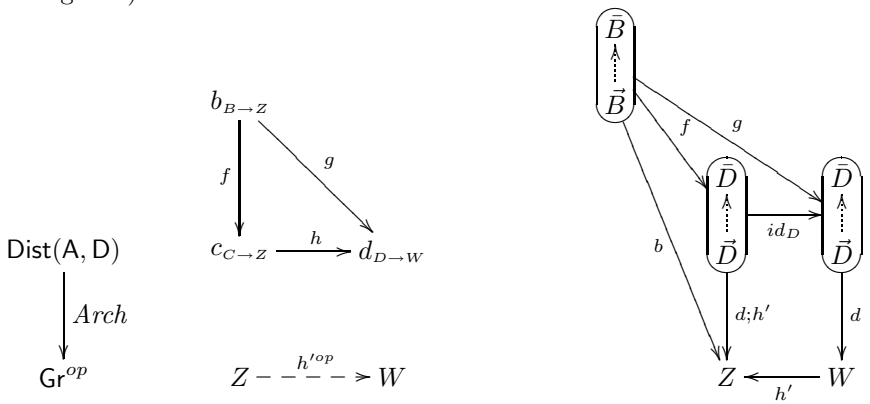
6 Given a fibration F with a cleavage cl, any $u : Y \to X$ in X determines a reindexing functor $u^* : F^{-1}(X) \to F^{-1}(Y)$ as follows:
— for an object C in $F^{-1}(X)$, $u^*(C)$ is the domain of $cl(u, C)$
— for a morphism $f : D \to C$ in $F^{-1}(X)$, $u^*(f)$ is the unique morphism in $F^{-1}(Y)$ making $u^*(f); cl(u, C) = cl(u, D); f$.

7 For every $Y \in Obj(\mathsf{X})$ there is a natural isomorphism $id_{F^{-1}(Y)} \rightrightarrows (id_Y)^*$, determined by the universal property of $cl(id_Y, A)$ for each $A \in Obj(\mathsf{A})$. For all $u : Y \to X$ and $w : X \to W$ there is a natural isomorphism $w^*; u^* \rightrightarrows (u; w)^*$, determined by $cl(u; w, A)$. If these isomorphisms are identities then the fibration is *split*.

□

Proposition 2.26. For any (A, D), $Arch : \mathsf{Dist}(\mathsf{A}, \mathsf{D}) \to \mathsf{Gr}^{op}$ is a (split) fibration.

Proof. The diagram to the left illustrates the fibration situation and the one to the right gives details for the present context (dashed arrows indicate the directions in the $_^{op}$ categories):



For an Gr^{op}-morphism $h'^{op} : Z \dashrightarrow W$ and distribution $d = d_{D \to W}$, the cleavage is defined

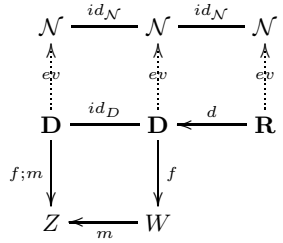


Fig. 12. The summary of the example.

as

$$cl(h'^{op}, d_{D\to W}) \stackrel{\text{def}}{=} h : c_{D\to Z} \to d_{D\to W} \text{ where } \begin{cases} h = \langle id_D, h' \rangle \\ c_{D\to Z} = d; h' \end{cases} \tag{6}$$

It is trivially a morphism in $Mor(\mathsf{Dist}(\mathsf{A}, \mathsf{D}))$ since $id_{\vec{Gr}(D)}; c = d; h'$.

Let $g : b_{B\to Z} \to d_{D\to W}$ be $\langle\langle \bar{g}, \vec{g}\rangle, h'\rangle$ – a distribution over h'^{op}. The unique $f : b \to c$ with $Arch(f) = id_Z$ making $f; h = g$ is then $\langle\langle \bar{g}, \vec{g}\rangle, id_Z\rangle$. $f \in Mor(\mathsf{Dist}(\mathsf{A}, \mathsf{D}))$ because $\vec{g}; b = d; h' = c; id_Z$, where the first equality holds since $g \in Mor(\mathsf{Dist}(\mathsf{A}, \mathsf{D}))$.

Composition of cleavages is a cleavage: for $h' : W \to Z$, $g' : Z \to V$ and $d_{D\to W} \in Arch^{-1}(W)$ we have $cl(h'^{op}, d_{D\to W}) = \langle id_D, h'\rangle$, $cl(g'^{op}, (d; h')_{D\to Z}) = \langle id_D, g'\rangle$, so
$cl(g'^{op}, (d; h')_{D\to Z}); cl(h'^{op}, d_{D\to W}) = \langle id_D, h'; g'\rangle = cl(g'^{op}; h'^{op}, d_{D\to W})$.
Hence $Arch$ is a fibration.

It is split: we have $(id_W)^*(d_{D\to W}) = d_{D\to W}$, so the natural transformation $id_{Arch^{-1}(W)} \rightrightarrows (id_W)^*$ is identity. Composition of cleavages gives a cleavage, so we get $(h'^{op})^*; (g'^{op})^* = (g'^{op}; h'^{op})^*$. □

Example 2.27 The collection of our examples so far illustrates this idea of development. In example 2.11 we designed a dependency **R** which was then refined to **D** by the morphism $d : \mathbf{R} \to \mathbf{D}$ – assuming standard algebra for natural numbers $\mathcal{N}$. The dependency **D** was then distributed on a vector W by $f_{D\to W}$ (Figure 8) and ported to Z (Figure 9) using the simulation $m : W \to Z$ from Figure 7. Provided that the simulation m and distribution f were available from a library, all the work we had to do was to design the mapping d. □

3. Constructing dependencies for recursive functions

So far we have considered relationships between various aspects – functional specification (algebra), dependencies (communication part) and distribution – given independently from each other. In this section we show how – under certain conditions – dependencies can be derived from specifications using grammars. Although the described construction may indicate the possibilities of reconsidering implementation techniques for functional languages, it should be emphasized that we are here concerned only with *a particular example* of utilizing computation algebra framework for desigining efficient implementations of algebraic specifications.

We recall that a *context free grammar* G is given by a 4-tuple $(T, N, \longrightarrow, S)$ where T is a set of terminal symbols, N is a set of non-terminal symbols auch that $N \cap T = \emptyset$, the set of production rules is $\longrightarrow \subseteq N \times (T \cup N)^*$, and $S \in N$ is the start symbol. From the grammar G we get the *derives* relation $\Longrightarrow \subseteq (N \cup T)^* \times (N \cup T)^*$ by $vXw \Longrightarrow vuw$ for strings $v, w \in (N \cup T)^*$ and productions $X \longrightarrow u$. The grammar defines the *language*, the set of strings, $\mathcal{L}(G) = \{w \in T^* | S \Longrightarrow^* w\}$ where $\Longrightarrow^*$ is the reflexive, transitive closure of $\Longrightarrow$.

In terms of Figure 11, we are at the transition point from A to $\mathsf{CAlg}(\mathsf{A}, \mathsf{D})$: we have obtained a specific algebra A and introduce a new recursive function to be computed over A. For this new function, we construct the dependency relatively to the algebra A. The difference from the general setting described in Section 2.5 is that we want to compute a new function over a given algebra and not construct a dependency for the whole algebra (cf. Remark 2.24). This is quite a common scenario in practice and we will give a few examples.

Let Σ be a signature, $S_1 ... S_n, S$ be some sorts from Σ, and $R : S_1 \times ... \times S_n \to S$ a new function symbol not in Σ. A *generalized Σ-term recurrence* for R is given by a set of equations of the form

$$R(\overline{\tau_0}(X)) = \Phi_{\overline{\tau_0}(X)}(\ X, R(\overline{\tau_1}(X)), R(\overline{\tau_2}(X)), \ldots, R(\overline{\tau_z}(X))\ ) \tag{7}$$

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 is typically given by a *set* of such equations, i.e., different equations for different (sequences of) term(s) $\overline{\tau_0}(X)$. (This form of generalized recurrence was studied in (Haveraaen 90; Haveraaen 93; Čyras and Haveraaen 95; Haveraaen and Søreide 98) as a definition schema for "constructive recursive functions".)

Given a recurrence (7) and a Σ-algebra A, we take as the set of nodes for the shape of the dependency, the domain of R in A, i.e., the cartesian product $V = S_1^A \times ... \times S_n^A$.

We now construct a collection of context free grammars $G_{\Sigma,V}$, which will define the data dependency. As the non-terminal symbols of the grammar, we take the set $N = \{N_v : v \in V\}$ of distinct symbols indexed by the nodes ($N_u = N_v \Rightarrow u = v$). Treating the non-terminals N_v as place-holders of type S, we may now define substitution rules from (7). For each variable assignment $\alpha : X \to A$, let $\alpha_i \in V$ denote the interpretation of $\overline{\tau_i}(X)$ under α, i.e., $\alpha_i = \overline{\tau_i}^A(\alpha(X)) = \langle t_{1,i}^A(\alpha(X)), ..., t_{n,i}^A(\alpha(X)) \rangle$. For every expression form $\overline{\tau_0}$ (from the set of equations (7)) and every assignment $\alpha : X \to A$, we get the production:

$$N_{\alpha_0} \longrightarrow_{G_{\Sigma,V}} \Phi_{\alpha_0}^A(\alpha(X), N_{\alpha_1}, N_{\alpha_2}, \ldots, N_{\alpha_z}) \tag{8}$$

Thus, for each node v, we obtain a node-specific grammar $G_{\Sigma,v} = \langle \mathcal{F}, N, \longrightarrow_{G_{\Sigma,V}}, N_v \rangle$, i.e., where terminals are the function symbols $\mathcal{F}$ from Σ, non-terminals are the N's, productions are given by (8), and the start symbol is N_v.

We define the shape of the dependency $\overset{G_{\Sigma,V}}{\rightarrowtail}$ by introducing an edge for all pairs $u, v \in V$, according to the rule:

$$u \overset{G_{\Sigma,V}}{\rightarrowtail} v \quad \Leftrightarrow \quad N_v \longrightarrow_{G_{\Sigma,V}} rN_u t, \ \text{ for some } r, t \text{ with } rN_u t \in \mathcal{T}_\Sigma(N). \tag{9}$$

This gives us a graph structure on V, and we can define a "compatible" *lab*eling function by requiring that

$$\forall v \in V : lab_{G_{\Sigma,V}}(v) \in \mathcal{L}(G_{\Sigma,v}) \tag{10}$$

where $\mathcal{L}(G_{\Sigma,v})$ is the language generated by the grammar $G_{\Sigma,v}$ of node v.

The result may not be a data dependency: the graph $\langle V, \overset{G_{\Sigma,V}}{\rightarrowtail} \rangle$ obtained from (9) may be cyclic, or else the labeling (10) may not be unique, problems often due to the recurrence (7) not being well-defined to begin with. The following conditions on $G_{\Sigma,V}$ ensure that the result will be a data dependency – a (simple) DAG with unique labeling.

Proposition 3.1. If a collection of node-specific grammars $G_{\Sigma,V}$ 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.

Proof. Point 2. implies that $\Gamma_{G_{\Sigma,V}}$ is acyclic. For every $v \in V$, the language $\mathcal{L}(G_{\Sigma,v})$ is a one-element set since by Point 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 $\mathcal{L}(G_{\Sigma,v})$. □

In each particular case of a given recurrence (7) and an algebra A, the well-definedness of the resulting dependency has to be checked, for instance, by verifying the conditions of the above proposition.

Another obvious fact following from the assumptions of Proposition 3.1 is that if $u \overset{G_{\Sigma,V}}{\rightarrowtail} v$ then $lab_{G_{\Sigma,V}}(u)$ is a subterm of $lab_{G_{\Sigma,V}}(v)$.

Remark. The described construction obviously subsumes primitive recursion for which we have $\Sigma = \langle Nat, \mathcal{F} \rangle$, the only model $\mathcal{N}$ being that of the naturals such that $\mathcal{F}$ are the basic primitive recursive operations on naturals. Given recursive equation schema for an R with $n+1$ variables

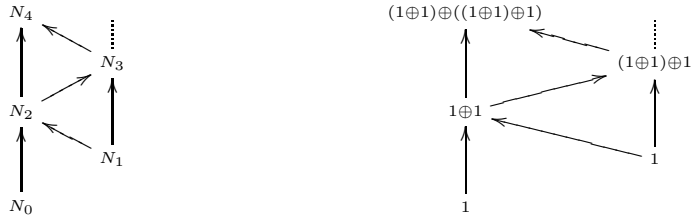
$$\begin{aligned} R(0, y_1, \ldots, y_n) &= g(y_1, \ldots, y_n) \\ R(\mathsf{s}(i), y_1, \ldots, y_n) &= h(i, y_1, \ldots, y_n, R(i, y_1, \ldots, y_n)), \end{aligned}$$

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

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

□

This approach to the recursive programming of functions is a derivative of the semantic (i.e. μ-recursive) approach to computability. Most functional languages, on the other hand, are compiled using (parallel) graph reduction techniques (Peyton and Simon 87) or, more generally, a syntax-oriented computability model based on term substitution, e.g. λ-calculus. Unlike these substitution based approaches, the construction described here leads often immediately to an efficient dependency structure which is amenable to



Γ_{Fib} with non-terminals Γ_{Fib} with labels

Fig. 13. The graph derived from the Fibonacci recurrence (11): on the left with the non-terminals, on the right with the language derived for each node (abbreviating $s(z)$ by 1).

a direct translation to efficient code on (parallel) machine hardware. The two following subsections illustrate this claim on two examples.

3.1. *A simple example – the Fibonacci function*

Take again the Fibonacci function from example 2.11, which we want to compute in a standard algebra $\mathcal{N}$ of natural numbers. That is, we have the signature $\Sigma = \langle\{Nat\}, \{s : Nat \to Nat, \oplus : Nat \times Nat \to Nat, z : Nat\}\rangle$, and $\mathcal{N}$ is given by $Nat^{\mathcal{N}} = \mathbb{N}$, the set of natural numbers, $s^{\mathcal{N}} =$ the successor function, $\oplus^{\mathcal{N}} =$ addition, and $z^{\mathcal{N}} = 0$.

The recursive specification of the Fibonacci function $F : Nat \to Nat$ is:

$$\begin{aligned} F(z) &= s(z) \\ F(s(z)) &= s(z) \\ F(i \oplus ss(z)) &= F(i) \oplus F(i \oplus s(z)) \end{aligned} \tag{11}$$

We choose $\mathbb{N}$ as the set of nodes V (corresponding to the arity of F), i.e., $V = \mathbb{N}$. With N_k, for $k \in \mathbb{N}$, as non-terminal symbols, we now obtain the production rules for the set of node-specific grammars (using that $z^{\mathcal{N}} = 0$, $s(z)^{\mathcal{N}} = 1$ and $ss(z)^{\mathcal{N}} = 2$ for the indices):

$$\begin{aligned} N_0 &\longrightarrow s(z) \\ N_1 &\longrightarrow s(z) \\ N_{i+2} &\longrightarrow (N_i \oplus N_{i+1}) \end{aligned} \tag{12}$$

Each non-terminal has only one production and so each node will be labeled by a unique ground term generated by the corresponding grammar. The resulting graph and labeling (with parentheses added for readability) are shown in Figure 13.

Notice that although labeling uses fully unfolded terms, the graph itself is not the usual tree of recursive calls as, e.g. **R** in Figure 3, but a much more efficient graph Γ_{Fib}.

3.2. *An example of parallelization - Fast Fourier Transform*

To illustrate the application to actual parallelization of programs, we review the technique of the fast Fourier Transform, FFT, due to (Cooley and Tukey 65). We first present the background problem and its solution given by a recurrence. The dependency for this recurrence obtained by our technique turns out to have the well known shape of the *butterfly graph*. We then show a standard distribution of the butterfly on a hypercube architecture.

3.2.1. *The numerical problem.* In many areas, computing of n-order polynomials $p(x) = \sum_{k=0}^{n} p_k x^k$, $0 \leq n$, 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 complex number ω such that $\omega^n = 1$. For every $n \geq 1$ there are n distinct complex roots, spread at n regular intervals along the unit circle in complex space, numbered counterclockwise starting at the number 1 which is root number 0. The j'th such root is denoted ω_n^j, and satisfies

$$\omega_n^0 = 1 \qquad \omega_n^{j+1} = \omega_n^j \omega_n^1 \qquad \omega_n^j = \omega_n^{j \bmod n}$$

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 $\omega_n^j = e^{(2\pi i/n)*j} = cos(2\pi j/n) + sin(2\pi j/n)i$, where $i = \omega_4^1$ is the imaginary unit, i.e., $i^2 = -1$.

Cooley and Tukey (Cooley and Tukey 65) discovered a very efficient algorithm, the FFT, 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 the polynomial $p(x)$ can be split into "even" p_e and "odd" p_o parts such that $p(x) = p_e(x^2) + x * p_o(x^2)$, where $p_e(x) = \sum_{k=0}^{2^{n-1}} p_{2k} * x^k$ and $p_o(x) = \sum_{k=0}^{2^{n-1}} p_{2k+1} * x^k$. Taking into account the 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

$$\begin{aligned} F(j,0) &= p_j \\ F(j,m+1) &= F(s_0(j,m+1),m) + \omega_{2^{m+1}}^j F(s_1(j,m+1),m) \end{aligned} \tag{13}$$

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 value of the polynomial at the j-th complex root of unity equals then $p(\omega_{2^M}^j) = F(j,M)$.

3.2.2. *Constructing a dependency.* Applying our strategy we start with the equations (13) defining the recurrence. We work with the algebra of natural and complex numbers,

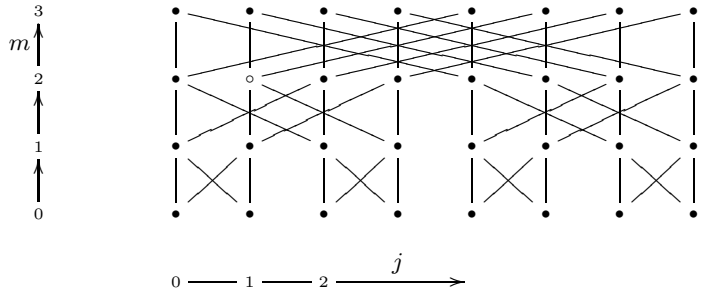


Fig. 14. The butterfly pattern of the recurrence F.

and as the set of nodes V we take the cartesian product $\{\langle j, m\rangle \in \mathbb{N} \times \mathbb{N} \mid 0 \leq j < 2^M, 0 \leq m \leq M\}$ – since F has two natural number arguments which are within these limits. The grammar resulting from the equations (13) is then (using $\oplus$ as the complex addition operation and $\otimes$ as the complex multiplication):

$$\begin{aligned} N_{\langle j,0\rangle} &\longrightarrow p_j \\ N_{\langle j,m+1\rangle} &\longrightarrow N_{\langle s_0(j,m+1),m\rangle} \oplus (\omega^j_{2^{m+1}} \otimes N_{\langle s_1(j,m+1),m\rangle}) \end{aligned} \tag{14}$$

with the coefficients p_j of the polynomial treated as input since the roots of unity $\omega^j_{2^{m+1}}$ are fixed once M is known. The resulting shape of the data dependency for $M = 3$ is given in Figure 14. Again, observe how this method leads directly to an "efficient" graph instead of generating the full tree of recursive calls.

Columns correspond to the j and rows to the m arguments with the leftmost column and bottom row numbered 0. The bottom row will be labeled with $p_0, p_1, p_2, ...$ according to the first kind of productions from (14). The labels in the upper rows will be generated using the second kind of productions from (14). We do not write it explicitly (lengthy labeling can be easily recovered from the grammar) but only observe that the label at node $\langle j, m\rangle$ will represent the value of $F(j, m)$. For instance, the node at row 2, column 1 (marked with the circle instead of bullet on the drawing) will represent the value of $F(1, 2)$.

3.2.3. *Distribution on a hypercube.* The shape from Figure 14 is known as a *butterfly graph*. Increasing M by one, i.e., taking $n = 2^{M+1}$, amounts to putting two butterflies of height M side by side, adding a new set of nodes on top (row $M + 1$), and connecting uppermost nodes (in the M'th row) of the one butterfly to its own and to the respective nodes of the other in the row $M + 1$.

Now, Figure 15 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 hypercube parallel program will then, on hypercube node j at time-step $m + 1$, compute $F(j, m + 1)$ using input data supplied by nodes $s_0(j, m + 1)$ and $s_1(j, m + 1)$ at time-step m. One of the data supplying nodes will be the same as the node j, the other will be its neighbor along dimension m.

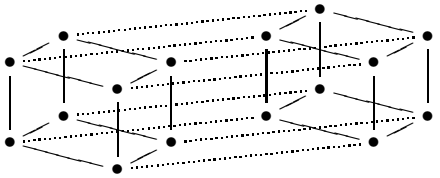


Fig. 15. Connections for hypercubes of dimension 3 (solid), and 4 (solid and dotted).

The Figure 16 shows a distribution of the butterfly dependency of height 3 from Figure 14 on the *st*-graph of the hypercube of dimension 3 from Figure 15. (Thicker arrows indicate the embedded edges of the butterfly; thinner lines the remaining edges of the hypercube.) Using the regularity in building a butterfly and a hypercube of higher dimensions, an analogous distribution is obtained for a butterfly of arbitrary height M into a hypercube of dimension M.

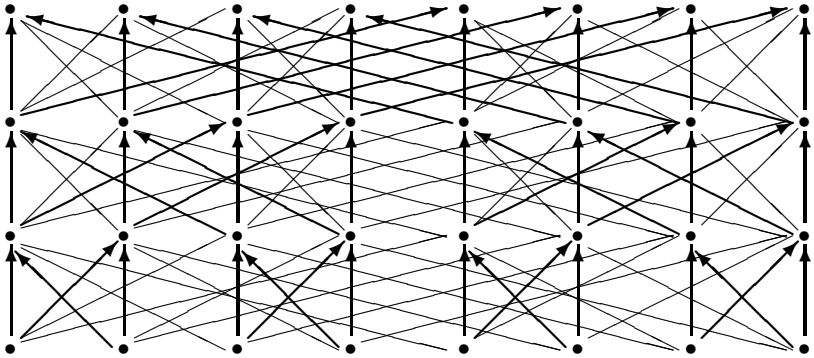


Fig. 16. Distribution of the butterfly on a hypercube.

4. More about development process

This section adds a few results concerning the development process using computation algebras. First, in Section 4.1, we show the counterpart of the standard result allowing one to view computation algebras over different signatures as objects of one category. Thus we obtain the means of refining the picture of the development process from Figure 11 by also admitting transitions between computation algebras over different signatures. Section 4.2 identifies then two subcategories of $\mathsf{CAlg}(\Sigma)$ with more specific morphisms which make the coupling of the algebra- and the communication-part tighter than in the general case of computation homomorphisms from Definition 2.13. The latter of these subcategories was used in the design of the actual specification formalism SAPPHIRE. Section 4.3 states two further properties of the category of computation algebras which are of relevance for the methodology of development.

4.1. *Computation Algebras over Different Signatures*

So far we have only considered computation algebras over one given signature. Thus, the development process illustrated in Figure 11 is not quite realistic since it is rather certain that such a process will involve change of signatures. We will now extend the standard way of flattening the categories of algebras indexed by signatures to the computation algebras and their distributions.

Let Sig be the category of signatures with injective signature morphisms. We write a dependency $\mathbf{D}$ as a triple $\langle I^D, \overset{D}{\rightarrowtail}, lab_{\mathbf{D}} \rangle$.

Definition 4.1 [**σ-reduct**] Let $\sigma : \Sigma' \to \Sigma \in Mor(\mathsf{Sig})$ and $\mathbf{D} \in \mathsf{DD}(\Sigma)$. The *$\sigma$-reduct* of $\mathbf{D}$, $\mathbf{D}|_\sigma \in \mathsf{DD}(\Sigma')$, is defined in two steps (we assume the ordinary extension of σ to $\sigma : \mathcal{T}_\Sigma \to \mathcal{T}_{\Sigma'}$):

1 let $\mathbf{X}$ be $\mathbf{D}$ with restricted re-labeling : $I^X = I^D$, $\overset{X}{\rightarrowtail} = \overset{D}{\rightarrowtail}$ and $lab_X(i) = \sigma^-(lab_{\mathbf{D}}(i))$, where $\sigma^- : \mathcal{T}_\Sigma \to \mathcal{T}_{\Sigma'}$ is given by $\sigma^-(t) = \begin{cases} t' & \text{such that } \sigma(t') = t \text{ if it exists} \\ \bot & \text{otherwise} \end{cases}$

2 $\mathbf{D}|_\sigma$ is then given by
— $I^{D|_\sigma} = \{i \in I^D : \sigma^-(lab_X(i)) \neq \bot\}$,
— $lab_{D|_\sigma}(i) = lab_X(i)$, and
— $i \overset{D|_\sigma}{\rightarrowtail} j \Leftrightarrow \begin{cases} i \overset{D}{\rightarrowtail} j \quad or \\ \exists \text{ path } i \overset{X}{\rightarrowtail} k_1 \cdots k_z \overset{X}{\rightarrowtail} j \text{ and } 1 \leq n \leq z \Rightarrow lab_X(k_n) = \bot \end{cases}$

□

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 unlabeled ($\bot$-labeled) intermediary nodes into edges between the labeled 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:

Proposition 4.2. $\sigma : \Sigma' \to \Sigma \in Mor(\mathsf{Sig})$ 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 labeling: each node i of $\vec{B}|_\sigma$ is sent to the node $\vec{h}(i)|_\sigma = \vec{h}(i)$, and edge e of $\vec{B}|_\sigma$ onto the edge (path/set of paths) $\vec{h}(e)|_\sigma$ in $\vec{A}|_\sigma$. This is a well defined $\mathsf{DD}(\Sigma')$-morphism: commutativity $\vec{h}; ev_A; \bar{h} = ev_B$ implies the corresponding commutativity in the reduct, since the latter operations are restrictions of their pre-images. □

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

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.

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 $\langle \sigma, f\rangle; \langle \rho, g\rangle = \langle \sigma; \rho, f; g|_\rho\rangle$.

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

4.2. *More specific dependency morphisms*

The class $\mathsf{CAlg}(\Sigma)$ provides the general framework for possible specializations. 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.

4.2.1. *Computation 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 labeled by s to ones labeled by t (nor vice versa), since such mappings are not compatible with the algebras in A. Thus, compatibility of a $\mathsf{DD}(\Sigma)$ morphism $h : \mathbf{C} \to \mathbf{D}$ wrt. an algebra A means that h does not effect a re-labeling which is inconsistent with valuation in A. More precisely:

Definition 4.3 Let $h : \mathbf{C} \to \mathbf{D} \in Mor(\mathsf{DD}(\Sigma))$ and $A \in \mathsf{A} \subseteq \mathsf{Alg}(\Sigma)$.

1 h is *A-compatible* iff $\forall i \in I^C : (lab_{\mathbf{D}}(h(i)))^A = (lab_{\mathbf{C}}(i))^A$;

2 h is *A-compatible* iff it is A-compatible for all $A \in Obj(\mathsf{A})$;

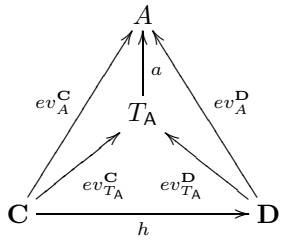
3 $\mathsf{DD}(\mathsf{A})$ is a wide subcategory of $\mathsf{DD}(\Sigma)$ with A-compatible morphisms.

□

Alternatively, $h : \mathbf{C} \to \mathbf{D}$ is A-compatible iff for each $A \in Obj(\mathsf{A})$, $\langle id_A, h\rangle$ is a computation homomorphism $\langle A, \mathbf{D}\rangle \to \langle A, \mathbf{C}\rangle$. We have the obvious

Proposition 4.4. Suppose that A has an initial object T_A, and let $h : \mathbf{C} \to \mathbf{D} \in Mor(\mathsf{DD}(\Sigma))$. Then: h is A-compatible $\Leftrightarrow$ h is T_A-compatible.

Proof. ($\Rightarrow$) is obvious. For ($\Leftarrow$) assume h is T_A-compatible. This means that the lower triangle of the following diagram commutes, $h; ev^{\mathbf{D}}_{T_\mathsf{A}} = ev^{\mathbf{C}}_{T_\mathsf{A}}$, where $ev^{\mathbf{Y}}_X$ assigns the values from an algebra X to the nodes of a dependency $\mathbf{Y}$ according to their labels (i.e. $ev^{\mathbf{Y}}_X(i) = (lab_{\mathbf{Y}}(i))^X$).



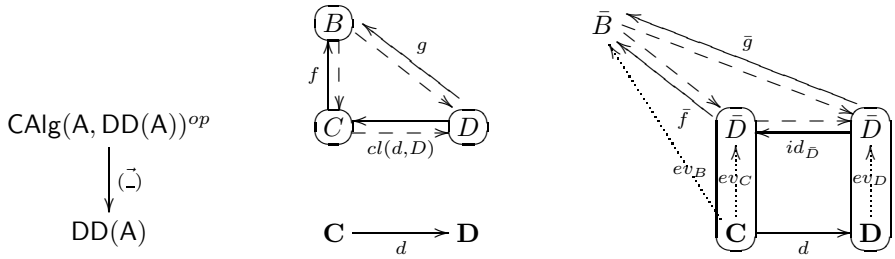
Since T_{A} is initial, for each $A \in \mathsf{A}$ and dependency $\mathbf{Y}$, $ev^{\mathbf{Y}}_A = ev^{\mathbf{Y}}_{T_{\mathsf{A}}}; a$ where a is the unique homomorphism $T_{\mathsf{A}} \to A$. But then $h; ev^{\mathbf{D}}_{T_{\mathsf{A}}} = ev^{\mathbf{C}}_{T_{\mathsf{A}}} \Rightarrow h; ev^{\mathbf{D}}_{T_{\mathsf{A}}}; a = ev^{\mathbf{C}}_{T_{\mathsf{A}}}; a \Rightarrow h; ev^{\mathbf{D}}_A = ev^{\mathbf{C}}_A$. □

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

For computation algebras with compatible dependency morphism, we also have the following fact about the restriction of the functor $(\vec{_})$ from (3):

Proposition 4.5. $(\vec{_}) : \mathsf{CAlg}(\mathsf{A}, \mathsf{DD}(\mathsf{A}))^{op} \to \mathsf{DD}(\mathsf{A})$ 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):



For a $\mathsf{DD}(\mathsf{A})$-morphism $d : \mathbf{C} \to \mathbf{D}$ and $D = \langle \bar{D}, \mathbf{D}, ev_D \rangle$, the cleavage is defined as

$$cl(d, D) \stackrel{\text{def}}{=} \langle id_{\bar{D}}, d \rangle : C \dashrightarrow D, \text{ where } C = \langle \bar{D}, \mathbf{C}, ev_C \rangle \tag{15}$$

with ev_C given by the last point of Definition 2.12. Since d is A-compatible, $d; ev_D; id_{\bar{D}} = ev_C$, and so the cleavage is a morphism in $\mathsf{CAlg}(\mathsf{A}, \mathsf{DD}(\mathsf{A}))^{op}$.

If $g : D \to B$ is over d ($g^{\vec{o}p} = d$), the unique $f : C \to B$ over $id_{\mathbf{C}}$ making $cl(d, D); f = g$ is $\langle \bar{g}, id_{\mathbf{C}} \rangle$ (i.e., $\langle id_{\bar{D}}, d \rangle; \langle \bar{g}, id_{\mathbf{C}} \rangle = \langle \bar{g}, d \rangle$).

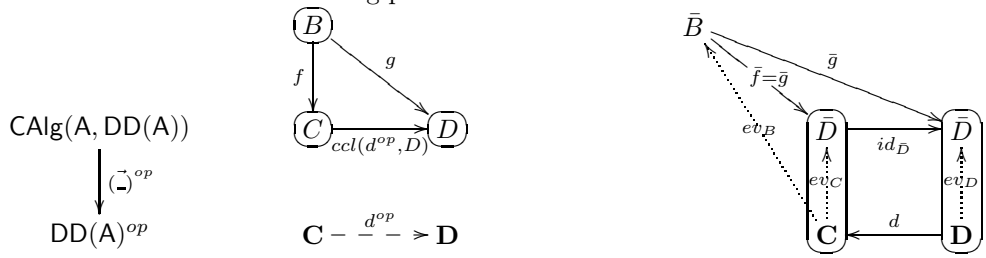
Obviously, the cleavages compose: for $c : \mathbf{B} \to \mathbf{C}$ and $d : \mathbf{C} \to \mathbf{D}$, we have $cl(d, D) = \langle id_{\bar{D}}, d \rangle$ and $cl(c, \langle \bar{D}, \mathbf{C} \rangle) = \langle id_{\bar{D}}, c \rangle$. So $cl(c, \langle \bar{D}, \mathbf{C} \rangle); cl(d, D) = \langle id_{\bar{D}}, c \rangle; \langle id_{\bar{D}}, d \rangle = \langle id_{\bar{D}}, c; d \rangle = cl(c; d, D)$. Hence $(\vec{_})$ is a fibration.

It is split: $(id_{\mathbf{D}})^*(\langle \bar{D}, \mathbf{D} \rangle) = \langle \bar{D}, \mathbf{D} \rangle$, so the natural transformation $id_{(\vec{\mathbf{D}})^{-1}} \rightrightarrows (id_{\mathbf{D}})^*$ is identity ($(\vec{\mathbf{D}})^{-1}$ denotes the fiber over $\mathbf{D}$). Also, since cleavages compose : $(d)^*; (c)^* = (c; d)^*$. □

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

Proposition 4.6. $(\vec{_}) : \mathsf{CAlg}(\mathsf{A}, \mathsf{DD}(\mathsf{A}))^{op} \to \mathsf{DD}(\mathsf{A})$ is a cofibration.

Proof. The proof and definition of cocleavage are essentially the same as in the Proposition 4.5. We have the following picture:



For a $\mathsf{DD}(\mathsf{A})^{op}$-morphism $d^{op} : \mathbf{C} \dashrightarrow \mathbf{D}$ and $D = \langle \bar{D}, \mathbf{D}, ev_D \rangle$, the cocleavage is defined as

$$ccl(d^{op}, D) \stackrel{\text{def}}{=} \langle id_{\bar{D}}, d \rangle : C \to D \text{ where } C = \langle \bar{D}, \mathbf{C}, ev_C \rangle, \tag{16}$$

with ev_C induced by $lab_{\mathbf{C}}$ according to Definition 2.12.3. Trivial repetition of the arguments from the proof of Proposition 4.5 yields the conclusion. □

This, together with the Proposition 4.5, means that $(\vec{_})$ is a bifibration and gives

Corollary 4.7. Reindexing functors induced by $((\vec{_}), cl)$ have left adjoints.

Finally, we may refer back to Section 4.1 and observe that, for an injective signature morphism $\sigma : \Sigma' \to \Sigma$, if $d : \mathbf{C} \to \mathbf{D} \in Mor(\mathsf{DD}(\Sigma))$ is A-compatible for an $A \in Obj(\mathsf{Alg}(\Sigma))$, then $d|_\sigma$ is $A|_\sigma$-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 $\mathsf{A}|_\sigma$-compatible. (In general, D may be only a subcategory of $\mathsf{DD}(\mathsf{A}|_\sigma)$.)

4.2.2. *Graph-morphisms.* We believe that refinement of data dependencies – at least as expressed by the compatible dependency morphisms in Definition 4.3 – 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 generality since data dependencies are quite concrete objects introduced towards the very end of the development process. In particular, passing from one computation algebra to another should not imply the introduction of new dependencies, only (and at most) purely syntactic memoisation. It is only the distribution morphisms which adjust the dependency graph to match it to the communication structure of a given architecture.

Accepting this view, one could work with a class of computation algebras (A, D) where $\mathsf{A} \subseteq \mathsf{Alg}(\Sigma)$ and the dependencies in D are restricted so that the target of the forgetful functor Gr projecting dependencies onto their graph-part (cf. (2) after Definition 2.10), is no longer the category Gr but its wide subcategory Gr°: morphisms in Gr° may map edges on paths but not on "parallel compositions" of paths. (Gr° can be defined as Gr in Definition 2.7 but with the adjunction $F^\circ \dashv U^\circ$ from Proposition 2.4 instead of $F^\oplus \dashv U^\oplus$.) Thus objects in D are simple DAGs and morphisms are the A-compatible dependency morphisms h such that $Gr(h)$ is a Gr°-morphism.

In fact, restricting our attention to merely syntactic memoisation, we restrict further

the morphisms in D to those which do not perform any relabeling but can, at most, identify some nodes from the source graph with identical labels. Thus, the morphisms are in fact $\mathsf{Alg}(\Sigma)$-compatible. Let us denote this class of computation algebras by $(\mathsf{A}, \mathsf{D}^\circ(\Sigma))$. Since $\mathsf{D}^\circ(\Sigma)$-morphisms do not perform any relabeling whatsoever, so in this class, any pair $h \in Mor(\mathsf{D}^\circ(\Sigma))$ and $g \in Mor(\mathsf{A})$ will give rise to a $(\mathsf{A}, \mathsf{D}^\circ(\Sigma))$-computation morphism.

Since $Mor(\mathsf{D}^\circ(\Sigma)) \subset Mor(\mathsf{DD}(\mathsf{A}))$, we can trivially repeat the constructions from Propositions 4.5 and 4.6, to see that $\vec{(_)} : (\mathsf{A}, \mathsf{D}^\circ(\Sigma))^{op} \to \mathsf{D}^\circ(\Sigma)$ is a bifibration.

The class $(\mathsf{A}, \mathsf{D}^\circ(\Sigma))$, 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 (Čyras and Haveraaen 95; Haveraaen and Søreide 98) (see section 3).

4.3. *Two more facts about* $\mathsf{CAlg}(\mathsf{A}, \mathsf{D})$.

The following two facts show further properties of $\mathsf{CAlg}(\mathsf{A}, \mathsf{D})$ which can be relevant in the development process. The one in Section 4.3.1 allows us to treat refinement along the algebra- and communication-part relatively independently from each other. This indicates the possibility that once developed chains of refinements of, say, dependency structures, could be reused in new contexts by coupling them with new actual algebras. The second one, Section 4.3.2, gives (some) conditions for the existence of initial computation algebras relatively to the exisitence of initial objects in A and D.

4.3.1. $\mathsf{CAlg}(\mathsf{A}, \mathsf{D})$ *as a double category.* A morphism between computation algebras consists of a pair of one standard algebra-morphism and one dependency-morphism (with additional compatibility criterion, Definition 2.13). Yet, the coupling of the two is loose enough to allow us to separate them in a development process. More precisely, any morphism in $\mathsf{CAlg}(\mathsf{A}, \mathsf{D})$ can be seen as applying an algebra morphism and then a dependency morphism (or vice versa). In a sense, $\mathsf{CAlg}(\mathsf{A}, \mathsf{D})$ "consists of" two categories: the "horizontal" one being A and the "vertical" one D.

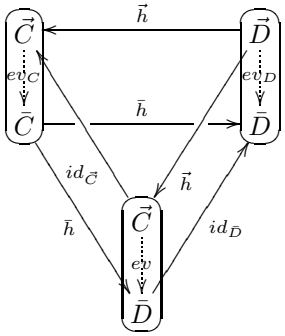
Expressing this relative independence categorically, we can show that computation algebras form a double category with horizontal and vertical arrows, respectively, of the form $\langle \bar{h}, id \rangle$ and $\langle id, \vec{h} \rangle$ (and cells being the commuting squares $\begin{array}{ccc} A & \xrightarrow{h} & B \\ {\scriptstyle v}\downarrow & & \downarrow{\scriptstyle u} \\ C & \xrightarrow{g} & D \end{array}$, where h, g are horizontal and u, v vertical arrows).

We will not spell out the details of this construction which is rather straightforward (for double categories, the reader may consult (Ehresmann 63; Mac Lane 71)). We only show the simple proposition illustrating that any morphism in $\mathsf{CAlg}(\mathsf{A}, \mathsf{D})$ can be seen as consisting of two, relatively independent steps: one mapping the algebra-part and another mapping the communication-part. 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 4.8. Each morphism $h : C \to D$ in $\mathsf{CAlg}(\mathsf{A}, \mathsf{D})$ can be factored so that

1 $h = \langle \bar{h}, id_{\vec{C}} \rangle ; \langle id_{\bar{D}}, \vec{h} \rangle$,
2 $h = \langle id_{\bar{C}}, \vec{h} \rangle ; \langle \bar{h}, id_{\vec{D}} \rangle$.

Proof. The following diagram details the first case:



The intermediary computation algebra is as shown in the Figure with $ev = ev_C ; \bar{h}$. This makes $\langle \bar{h}, id_{\vec{C}} \rangle$ a computation homomorphism $\langle \bar{C}, \vec{C}, ev_C \rangle \to \langle \bar{D}, \vec{C}, ev \rangle$. Also, since h is a computation homomorphism, we have for all $i \in I^{\vec{D}}$: $\bar{h}(ev_C(\vec{h}(i))) = ev_D(i)$, i.e., $ev(\vec{h}(i)) = ev_D(i)$ which means that $\langle id_{\bar{D}}, \vec{h} \rangle$ is a computation 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 $\langle \bar{C}, \vec{D}, \vec{h}; ev_C \rangle$. □

4.3.2. *Initial objects.* Define functor $\overleftarrow{(_)} : \mathsf{CAlg}(\mathsf{A}, \mathsf{D}) \to \mathsf{D}^{op}$ analogously to $\vec{(_)}$ in (3), i.e., $\overleftarrow{A} = \vec{A}$ and $\overleftarrow{\langle f, g \rangle} = g$ (the communication part g of a morphism $\langle h, g \rangle$ in $\mathsf{CAlg}(\mathsf{A}, \mathsf{D})$ is contravariant to the whole morphism). We then have the following general fact:

Proposition 4.9. If A has an initial object T_{A}, then the functor $F : \mathsf{D}^{op} \to \mathsf{CAlg}(\mathsf{A}, \mathsf{D})$ defined by $F(\mathbf{C}) = \langle T_{\mathsf{A}}, \mathbf{C} \rangle$ and $F(g) = \langle id_{T_{\mathsf{A}}}, g \rangle$ is left adjoint to $\overleftarrow{(_)}$, $F \dashv \overleftarrow{(_)}$, with identity as unit.

Thus, if D^{op} has an initial object, so does $\mathsf{CAlg}(\mathsf{A}, \mathsf{D})$. However, in general, this will not 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 are 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 standard algebras as computation algebras with an empty dependency.

Nevertheless, the proposition gives us useful information in a slightly more restricted context. Typically, initial object in a model class A of a specification is obtained as (or from) a term model. Then, given a particular (non-empty) dependency, the proposition provides an "initial" evaluation strategy over such a term model.

5. Nondeterminism

Since parallelism and distributed programs often are a source of nondeterminism, we suggest here how this phenomenon can be incorporated into the present framework. There

are at least two ways to do it. The first, in Section 5.1, utilizes the notion of multialgebras, which are designed for incorporating nondeterminism into algebraic specification as an abstraction mechanism (Husßmann 93; Walicki and Meldal 94; Walicki and Meldal 97). The second, Section 5.2, is biased towards the operational view, according to which nondeterminism is something which arises only during the actual computations.

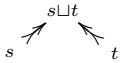
5.1. *The multialgebraic view*

Revisiting the Definition 2.12 of computation algebras and the subsequent constructions, one can observe that we have not made any significant assumptions about the underlying category of Σ-algebras. The only essential requirement was that ev-function assigns values from an algebra to each node of the dependency in a way respecting the labeling of the node. In fact, instead of the usual Σ-algebras, we can work with a different category for $\mathsf{Alg}(\Sigma)$.

For modeling nondeterminism, a natural choice is (some) category of multialgebras with multihomomorphisms, (Walicki and Białasik 97). 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 of operations is defined pointwise. A (ground) term is interpreted in a multialgebra as the set (of its possible results). The notion of a homomorphism $h : A \to B$ may be generalized in various ways ((Walicki and Białasik 97) surveys the possibilities), but the most common one replaces the usual homomorphism condition by the requirement $h(f(x)^A) \subseteq f^B(h(x))$, for each function symbol $f \in \Sigma$. There is also a choice of the primitive operations for writing specifications. The most common approaches use a primitive predicate, $s \prec t$ (interpreted in an algebra A as set inclusion $s^A \subseteq t^A$, i.e., as t being at least as nondeterministic as s), and/or the nondeterministic binary choice $_ \sqcup _ : S \times S \to S$, for some/every sort S. We do not discuss the specification languages here, and in the examples to follow we will use the above, most common, definition of multihomomorphism.

The change of the category of algebras is the only modification needed. The computation homomorphism h (with $\bar{h}$ being a multihomomorphism) allows us, for instance, to refine an algebra B to a more deterministic algebra A as illustrated in Figure 17. All operations except $\sqcup$ are deterministic. $\vec{h}$ may relabel $s \sqcup t$ to s because both terms have the same sort and, furthermore, because $\bar{A}$ is more deterministic than $\bar{B}$: for $i \in I^B$ with $lab(i) = s \sqcup t$, we have that $\bar{h}(ev_A(\vec{h}(i))) = \bar{h}(s^A) = s^B \subseteq s^B \cup t^B = (s \sqcup t)^B = ev_B(i)$, i.e., $\bar{h}$ is a multihomomorphism.

5.2. *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. The dependency  from Figure 17 can be read

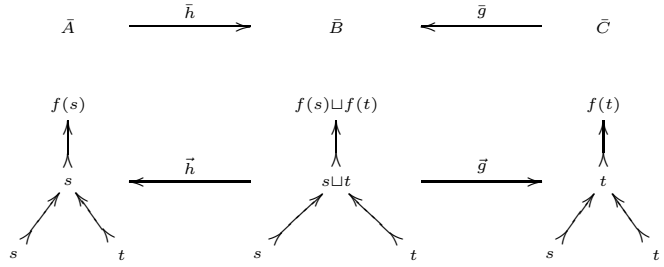


Fig. 17.

as expressing just that: the value at the topmost node depends on the values at the other two nodes – it will simply be the one arriving first. Multialgebras provide an abstract model for this kind of phenomena. We will now sketch a more detailed, low-level way of including them into computation algebra framework.

The computation related information for a computation algebra $A = \langle \bar{A}, \vec{A} \rangle$ is represented by its communication part. Thus we may retain the standard notion of algebra $\bar{A}$ and try to model nondeterminism in $\vec{A}$. We simply allow for labeling the nodes of $\vec{A}$ by *sets of terms*, rather than unique terms. The set labeling a node represents the possible values computed at the node. The source of the evaluation function ev_A is then a set of pairs $\langle i, t \rangle$ where i is some node and t one of the terms labeling $i : \bigcup_{i \in I^{\vec{Gr}(A)}} \{\langle i, t \rangle : t \in lab^A(i)\}$. 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 labeling merely indicates the range of possibilities.

Definition 5.1 $\mathsf{nDD}(\Sigma)$ is the category where (Γ is as in Definition 2.1):

1 objects are Σ n-data dependencies – pairs $\mathbf{G} = \langle G, lab \rangle$ where $G \in Obj(\mathsf{Gr})$ is a simple DAG and $lab : I^G \to \mathcal{P}^+(\mathcal{T}_\Sigma)$ is a function labeling each node of this graph with a non-empty set of ground Σ-terms of the same sort, i.e., $\forall i \; \forall s, t \in lab(i) : Sort(s) = Sort(t)$;

2 an $\mathsf{nDD}(\Sigma)$-morphism $h : \mathbf{B} \to \mathbf{A}$ is a pair $\langle \vec{h}, h_{lab} \rangle$ where
   — $\vec{h}$ is a (sort compatible) Gr-morphism $\vec{h} : Gr(\mathbf{B}) \to Gr(\mathbf{A})$,
   — h_{lab} is a family $\{h_i : lab_{\mathbf{A}}(\vec{h}(i)) \to lab_{\mathbf{B}}(i)\}_{i \in I^B}$.

□

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 h_{lab} mapping (which we can treat as a partial function $I^A \times \mathcal{T}_\Sigma \to \mathcal{T}_\Sigma$ with the domain given by all the pairs $\langle \vec{h}(i), t \rangle$ for all $i \in I^B$ with $t \in lab_{\mathbf{A}}(\vec{h}(i))$ picks the subset of labels of $lab_{\mathbf{B}}(i)$ which is the pre-image of the label set $lab_{\mathbf{A}}(\vec{h}(i))$. For instance, in Figure 18, $\mathbf{B}$ is a dependency for nondeterministic choice between s and t which is then propagated as the argument to the function f. (The relabeling is merely a restriction and h_{lab} is the obvious inclusion.)

The node i in $\mathbf{B}$ 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 deterministic refinement of $\mathbf{B}$ which always chooses s and computes only $f(s)$ is given in $\mathbf{A}$ with the morphism h.

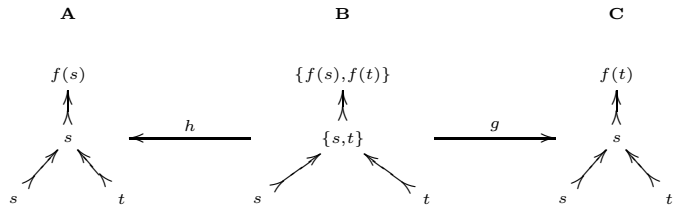


Fig. 18.

Observe, however, that the definitions of an n-dependency and an $\mathsf{nDD}(\Sigma)$-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 morphism only captures the preservation of dependencies (in terms of the underlying graphs) and the non-increasing 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 5.2 Given $\mathsf{A} \subseteq \mathsf{Alg}(\Sigma)$ and $\mathsf{D} \subseteq \mathsf{nDD}(\Sigma)$; an (A, D) *n-computation 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) function such that $\forall t \in lab_{\vec{C}}(i) : ev_C(i,t) = t^{\bar{C}}$.

□

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

Definition 5.3 An (A, D) *n-computation homomorphism* $h : C \to D$ is $\langle \bar{h}, \langle \vec{h}, h_{lab} \rangle \rangle$ where

1 $\bar{h} : \bar{C} \to \bar{D} \in Mor(\mathsf{A})$;
2 $\langle \vec{h}, h_{lab} \rangle : \vec{D} \to \vec{C} \in Mor(\mathsf{D})$, and
3 for each $i \in I^{\vec{G}r(D)}$ and $t \in lab_{\mathbf{C}}(\vec{h}(i))$, we have $\bar{h}(ev_{\mathbf{C}}(\vec{h}(i), t)) = ev_{\mathbf{D}}(i, h_{lab}(\vec{h}(i), t))$.

□

The last condition (illustrated in Figure 19) is a generalization of the commutativity condition for the computation homomorphisms from the Definition 2.13. In the deterministic case, the two coincide since then h_{lab} is uniquely determined by $\vec{h}$.

The category $\mathsf{nCAlg}(\mathsf{A}, \mathsf{D})$ has (A, D) n-computation algebras as objects and (A, D) n-computation homomorphisms as morphisms. Distributions of $\mathsf{nCAlg}(\mathsf{A}, \mathsf{D})$ are defined as before (Definition 2.19), and so are the projection functors $(\vec{_})$ and *Arch*. The Proposition 2.26 generalizes trivially to the nondeterministic context by an entirely analogous proof using the construction (6). Modifying the first point of Definition 4.3 to the requirement $\forall i \in I^{Gr(\mathbf{C})}, t \in lab_{\mathbf{D}}(\vec{h}(i)) : t^A = (h_{lab}(\vec{h}(i), t))^A$, we may define the class $\mathsf{nCAlg}(\mathsf{A}, \mathsf{nDD}(\mathsf{A}))$ of n-algebras with compatible dependency morphisms. The Lemma

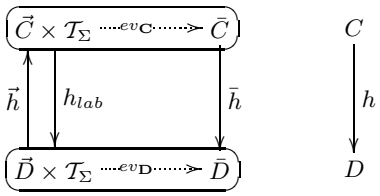


Fig. 19. An n-computation homomorphism.

4.5 generalizes then equally easy to this class by a construction analogous to (16). We therefore do not do it explicitly here but only emphasize that the earlier results remain valid when we put n-'s at all relevant places.

6. Conclusions and further work

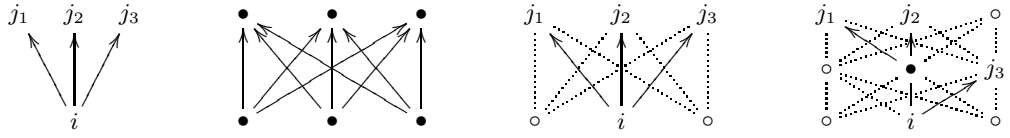
We have introduced a general notion of computation algebras which extends the standard algebraic semantics with additional structures, data dependencies, carrying information concerning the evaluation strategy. The dependency morphisms can be used to design more efficient implementations, in particular, by syntactic memoisation. The combination of dependencies with actual algebras and the notion of computation 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. Dependency and distribution morphisms express, respectively, two aspects of parallelization: (a) the logical dependency decomposition and (b) partitioning and routing on a physical medium. We have shown how indexing computation algebras with machine architectures yields a fibration. Its explicitly defined cleavage provides a way of porting distributions between various (parallel) architectures.

The general framework admits the definition of various subclasses of computation 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.

Extending algebra(s) with the computation and control information resembles the otherwise widely studied attempts of adding such an information to term rewriting systems (Clavel *et al.* 96; Borovansky *et al.* 98a; Borovansky *et al.* 98b; Clavel *et al.* 99; Visser 99). In this respect the main difference is that we are not considering executable specifications (like term rewriting systems) and their efficiency or capabilities for modeling computations, but efficiency of an actual implementation of a (low level, detailed) specification. In particular, dependencies can be adjusted to actual (parallel) machine architectures which are included as a part of the overall picture in our model of development process.

In Section 3 we have suggested a method allowing one to construct data dependency from a given specification of recursive function to be computed in a given algebra. A special case of this method, combined with a subcategory $(\mathsf{A}, \mathsf{D}^\circ(\Sigma))$ of computation algebras (cf. Section 4.2.2) and embedded in the programming language SAPPHIRE, has been worked out in more detail and used in practical applications for parallelization of programs (Haveraaen 93; Čyras and Haveraaen 95; Haveraaen and Søreide 98; Haver-



D *B and S* **D** → *B* **D** → *S*

a) b) c) d)

Fig. 20. a) Data dependency **D**, b) space-time representation of a broadcast, B, and a stepwise, S, architecture, c) distribution of **D** on B, and d) on S.

aaen 99). This construction of efficient data dependencies for recursive functions leads towards consideration of efficient implementations of functional programming languages, in particular, of basing such implementations on the semantic (μ-recursive), rather then syntactic (substitution oriented) computational model. This, however, is an open question for future research which falls outside the scope of the present paper.

Open issues

Computation algebras offer a mathematical framework for modeling efficient implementation of algebras. Application of this framework to actual architectures and (parallel) programming languages is the main topic deserving further study. We mention here three issues of more detailed character.

1. Space-time representation does not capture all aspects of parallel architectures. For instance, the simulation m of 2-dimensional vector on a 3-processor ring from Figure 7.c) assumes that multiple data can be sent along the channels. (The third node in the rightmost column in Figure 7.c) receives two pieces of data which *both* have to be passed to the node above it.) In general, this may be precluded on machines where sending multiple data requires several computation steps. Thus actual simulation morphisms have to respect such additional restrictions.

Similar issue concerns the kind of allowed communications. Figure 20.b) shows identical space-time representation of two different architectures: B – with broadcats, and S with stepwise communication (i.e., a processor can communicate only along one channel at a time). The distribution c) of **D** on B is not meaningful on S which can be achieved in two steps as illustrated in d).

Distribution morphisms can distinguish between these situations but since this information is not part of the st-representation, this is something that has to be done when designing actual morphisms. One should verify that space-time graphs, together with the simulation and distribution morphisms, offer sufficient representational power.

2. Portability vs. Efficiency. Another issue concerns the efficiency of the ported implementations (distributions). For instance, the implementation achieved in Figure 9.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 $(Z_3, 1)$ – holding $F(1)$ – to $(Z_3, 2)$. This delay occured because we have used a *generic* simulation m of W by Z (from Figure 7.c), i.e., one that allows us to port *any* implementation from W to Z. Even if such a simulation is generically optimal (i.e., uses minimum time on the target architecture for all possible simulations from the source architecture), for a particular case it may be possible to find a more efficient mapping. However, though efficiency of computation always is a concern, general portability and re-usability of software is at least equally important. 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 successfully ported an implementation, there is nothing precluding some final optimization in order to achieve a more efficient product.

3. Code generation. The most important issue concerns generation of actual code. 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. 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 which is determined at run-time). A unifying view is to think of the nodes (in a dependency) 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. 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 distribution (Definition 2.19) does capture several aspects of parallelization. Firstly, different "columns" in the *st*-graph (the sets of nodes $\{\langle s, t\rangle : t \in \mathbb{N}\}$ for different s) of a given architecture will represent different processors and so a distribution mapping amounts to *partitioning* of the code onto these processors. Furthermore, the links between different components ("columns") 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.) Finally, distribution morphisms represent (part of) 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 labeling 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;`", where X, Y are the pieces of code resulting from the labeling of i and j. (The directions of sending/receiving will often be identified not by the (name of) another processor but by the local (name of a) channel. This is another specific distinction, depending on the actual machine and programming language, which we did not address at all.)

Our model comprises only the abstract notion, not such a concrete view of implementation. Code generation for a special class of computation algebras specified equationally with the explicit specification of the dependency structures was studied in connection with the SAPPHIRE methodology for development of parallel programs, (Haveraaen 93; Haveraaen and Søreide 98).

Other questions for further research.

— Although we did not discuss any specification formalism, it should be obvious that all the aspects of our setting are amenable to a description using algebraic techniques. In particular, data dependencies, architectures, and morphisms between them are standard algebraic objects. Future research should integrate description of data dependencies into an actual specification framework.

— An interesting alternative would be to investigate the possibilities of extracting data dependencies from a given algebraic specification. Section 3 opens a way in this direction. Open questions concern, in particular, the syntactic conditions on the recursive definitions such as (7) which would ensure well-definedness of the resulting function in specific algebras. General construction of graphs and labelings from specifications is a broader issue worth closer investigation.

— A variety of dependency and, especially, simulation morphisms between existing architectures could be 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 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.

— We would like to make a closer study of 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 this purpose.

Acknowledgments

We thank Andrzej Tarlecki and Eric Wagner for interesting suggestions and comments on earlier versions of this paper. An anonymous referee suggested to extend the definition of DAG°-morphisms by equipping graphs with the "parallel composition" using Kleisli construction. We also thank another anonymous referee for suggesting improvements of the structure of the paper. Kristoffer Rose deserves thanks for designing and making available the `XY-pic` package for drawing nice diagrams.

References

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

P. Borovansky, C. Kirchner, H. Kirchner. A functional view of rewriting and strategies for a semantics of ELAN. In *The Third Fuji International Symposium on Functional and Logic Programming*, pp. 143-167, Kyoto (Japan), World Scientific, (1998).

P. Borovansky, C. Kirchner, H. Kirchner. Rewriting as a Unified Specification Tool for Logic and Control: The ELAN Language. In *Second International Workshop on the Theory and Practice of Algebraic Specifications ASF+SDF'97*, Electronic Workshops in Computing, eWiC web site: http://ewic.springer.co.uk/, Amsterdam (The Netherlands), Springer, (1998).

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

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

M. Clavel, F. Durán, S. Eker, J. Meseguer, M.-O. Stehr. Maude as a Formal Meta-Tool. In the *Proceedings of FM'99*, The World Congress On Formal Methods In The Development Of Computing Systems, Toulouse (France), (1999).

M. Clavel, S. Eker, P. Lincoln and J. Meseguer. Principles of Maude. In Proc. 1st Intl. Workshop on Rewriting Logic and its Applications, Electronic Notes in Theoretical Computer Science, Elsevier Sciences, (1996).

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

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

C. Ehresmann. *Catégories Structurées,* Ann.Sci.Ecole Norm.Sup. 80, (1963).

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

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

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).

M. Haveraaen: An algebra of data dependencies and embeddings for parallel programming. To appear in *Formal Aspects of Computing*, 1999.

M. Haveraaen, S. Søreide: *Solving recursive problems in linear time using Constructive Recursion*. Proceedings of *Norsk Informatikk Konferanse NIK'98*, Tapir, Norway, 1998, pp. 310–321.

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

H. Hußmann. *Nondeterminism in Algebraic Specifications and Algebraic Programs.* Birkhäuser, (1993).

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

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

S. Mac Lane. *Categories for the Working Mathematician*, Springer, (1971).

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

J. Peyton, L. Simon. *The Implementation of Functional Programming Languages*, Prentice-Hall, (1987).

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

E. Visser. Strategic Pattern Matching, in *Rewriting Techniques and Applications (RTA'99)*, LNCS vol. 1631, pp. 30-44, Springer, (1999).

M. Walicki, M. Białasik. Categories of Relational Structures. In *Recent Trends in Data Type Specification,* LNCS, vol. 1376, (1997).

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

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).

M. Walicki, S. Meldal. Algebraic Approaches to Nondeterminism – an Overview. *ACM Computing Surveys,* 29, 1, (1997).