Control-Flow Semantics for Assembly-Level Data-Flow Graphs

Wolfram Kahl*, Christopher K. Anand, and Jacques Carette

SQRL, McMaster University, Hamilton, Ontario, Canada

Abstract. As part of a larger project, we have built a declarative assembly language that enables us to specify multiple code paths to compute particular quantities, giving the instruction scheduler more flexibility in balancing execution resources for superscalar execution.
Since the key design points for this language are to only describe data flow, have built-in facilities for redundancies, and still have code that *looks like* assembler, by virtue of consisting mainly of assembly instructions, we are basing the theoretical foundations on data-flow graph theory, and have to accommodate also relational aspects.
Using functorial semantics into a Kleene category of "hyper-paths", we formally capture the *data-flow-with-choice* aspects of this language and its implementation, providing also the framework for the necessary correctness proofs.

1 Introduction

Magnetic resonance imaging (MRI) relies on highly efficient signal processing software — for example in medical applications, higher efficiency can make quite an important qualitative difference. The state of the art in the development of such software is that a scientist starts from a mathematical model and produces an appropriate signal processing algorithm; as first step towards an implementation this algorithm is directly translated into a prototype program, with reasonable confidence in its correctness. This is then turned over to a "digital signal processing guru" who will apply — manually! — different kinds of code transformations and optimisations, up to manual rearrangement of assembly code.

It is obvious that it is not easy to be fully confident in the correctness of software coming out of such a process. However, particularly in medical applications, correctness can be crucial, and defects in MRI signal processing software could manifest themselves as visible artifacts in the generated images that could introduce problems in the medical uses of these images.

The Coconut project prepares to produce a system that provides a coherent and consistent path from a mathematical specification of signal processing problems to verified *and* highly optimised machine code [2]. As part of this project, we encountered a need for a language of a quite peculiar nature: we needed to specify choices amongst different "equivalent" computation paths made up

* This research has been supported by an NSERC Discovery Grant

of low-level assembler. An intelligent instruction scheduler will choose the best path, using built-in knowledge of the intricacies of a modern, vectorised and pipelined CPU architecture. Our collective experience told us that we should be specifying our problem in a declarative manner, to give maximal freedom to the scheduler. We decided to see if we could get these rather different paradigms (declarative and assembly) to coexist, and serve as the main language for our compiler's back end.

The central idea is that this approach should allow us a separation of concerns in the code generation part of our special-purpose compiler:

- The *(assembly) code generator* will use knowledge of the *mathematical semantics* of assembly instructions to generate *correct* assembly code, but it will leave control flow decisions open as far as possible.
- The *scheduler* (or *assembler*) uses knowledge about the *resource consumption* of assembly instructions to generate *fast* machine code; correctness is guaranteed by the fact that the scheduler essentially performs only a selection of one path among those proposed by the code generator.

Our intermediate *declarative assembly language* therefore represents the semantics of its programs mostly as data flow; it allows to express some control flow constraints, but essentially leaves all efficiency-related instruction selection and scheduling decisions open.

Our targets are vectorised and pipelined CPUs, currently PowerPC 745X and 970, that are commonly used in signal processing applications. We design the scheduler (with appropriate support from the code generation component) to be able to automate a number of "tricks" used in manual optimisation. For example, it will be able to take advantage of limited precision requirements, and more generally, choose between "equivalent" machine code computations, which includes choosing instructions that produce the same results with different resource consumption, and choosing between computations that produce different intermediate values which can be used interchangeably. Other tricks avoid register spill for example via recomputation of previously available values, or via the use of renaming registers (used internally by some PowerPC versions) as non-addressable intermediate storage.

These requirements motivated our decision to base our declarative assembly language essentially on data flow graphs, and to add choices of computation paths as a new feature.

Therefore, branches are eliminated from our declarative assembly language, and we express all control flow which cannot be eliminated by use of permutation and selection in special-purpose 'combinators' (not considered in this paper). Only the non-branching instructions (of the PowerPC 745X and PowerPC 970) can be used as labels in our *code graphs*, which we understand to be the abstract syntax of our assembly language.

2 Code Graphs

Term graphs are usually represented by graphs where nodes are labelled with function symbols and edges connect function calls with their arguments [12]. An alternative representation was introduced with the name of *jungle* by Hoffmann and Plump [7] for the purpose of efficient implementation of term rewriting systems (it is called "term graph" in [11]).

A *jungle* is a directed hypergraph where nodes are only labelled with type information (if applicable), function names are hyperedge labels, each hyperedge has a sequence of input tentacles and exactly one output tentacle, and for each node, there is at most one hyperedge that has its output tentacle incident with that node.

For representing our declarative assembly code fragments, we use a generalisation of the jungle concept, corresponding to Ştefănescu's "flow graphs" [14]:

Definition 2.1 A *code graph* $G = (\mathcal{N}, \mathcal{E}, \mathsf{In}, \mathsf{Out}, \mathsf{src}, \mathsf{trg}, \mathsf{eLab})$ over an edge label set ELab consists of

- a set $\mathcal{N}$ of *nodes* and a set $\mathcal{E}$ of *hyperedges* (or *edges*),
- two node sequences $\mathsf{In}, \mathsf{Out} : \mathcal{N}^*$ containing the *input nodes* and *output nodes* of the code graph,
- two functions $\mathsf{src}, \mathsf{trg} : \mathcal{E} \to \mathcal{N}^*$ assigning each hyperedge the sequence of its *source nodes* and *target nodes* respectively, and
- a function $\mathsf{eLab} : \mathcal{E} \to \mathsf{ELab}$ assigning each hyperedge its *edge label*, where the label has to be compatible with the numbers of source and target nodes of the edge. □

In Coconut, nodes are actually labelled with *types*, but this is not relevant for the current paper. Edge labels are either opcodes or constants, as can be seen in the example code graph above, where output tentacles are arrows from hyper-edges to nodes, and input tentacles are arrows from nodes to hyperedges — the ordering relation between in- resp. output tentacles incident with the same hyperedge is not made explicit in the drawing, but is part of the graph structure.

Acyclic code graphs where all edges have exactly one target node and no node is the target of several edges correspond to the jungles of [7] (called "term graphs" in [11]), which are essentially a hypergraph version of conventional term graphs. Since some operations produce more than one result, our hyperedges can have multiple output tentacles just as the primitives of Ştefănescu's flow graphs [14]; this also corresponds to the use of "hypersignatures" in [3]. In the application to PowerPC, the second result is always a condition code, i.e. carry or overflow, but we think it will be better to treat all results uniformly.

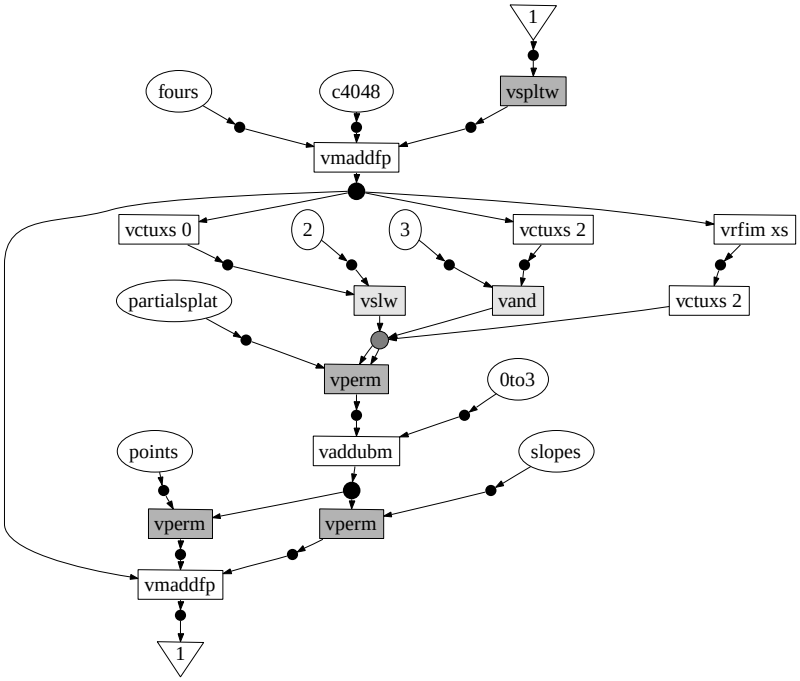
The more radical departure from conventional term-graph formalisms is that we allow *several* output tentacles to be incident with any one node — such *joining* tentacles are used for results that can be obtained in different ways, and also for situations where different intermediate values could be used interchangeably. In Ştefănescu's flownomials there are joins, too — Ştefănescu proposes two

interpretations for the "branch/join" pair of operations in data flow networks, one as "copy/equality test" and one as "split/merge" [13]. Our use is closer to a control-flow interpretation of join; the following list of typical applications of joins shows how this single feature opens up a large bag of "tricks" for code generation from code graphs:

- **Multiple entry points:** Many common mathematical functions are implemented (for the sake of efficiency) via algorithms with extra preconditions, and initial code ensures that those preconditions are satisfied. For example,
  - trigonometric functions are only calculated on a fundamental domain, and modulo calculations are first performed to put arguments into the fundamental domain;
  - some functions have a standard interface (e.g. choice of units), but an alternative interface is much more efficient, so the initial statements perform the necessary conversions.

  In both cases, we can eliminate the respective initial instructions if we can verify the stricter preconditions, or if we can rewrite the upstream calculation to produce results matched to the more efficient interface.
- **Instruction selection:** For example merging disjoint bit-fields by either logical `or` or arithmetic `add` instructions uses different functional units on some processors. In such cases, conventional optimising compilers switch instructions to get better schedules; in our approach, we emit a join in the assembly code and let the code graph scheduler select the better branch in each case.
- **Multiple code paths** (beyond single-instruction alternatives): It is possible to do some computations in different units, e.g., evaluating polynomials in the scalar floating point unit or the vector floating point unit. Such alternative code paths can be used in two ways:
  - `map f`, where `f` is a simple function, can be unrolled and performed simultaneously on different data in different execution units;
  - the code can be in-lined in different contexts where relative demand on execution units, register pressure, etc., vary enough to make one code path more efficient than another.

As an example, the following code graph arises in an implementation of a non-uniform Fourier transform — we always show code graphs with the sequences of input and output nodes indicated by arrows from, respectively to, numbered triangles; here, there is only one input and one output. Rectangles are instruction hyperedges, and ellipses are (zero-input) constant hyperedges. The light grey instructions require the vector integer unit; the dark grey instructions require the vector permutation unit, and all other instructions the vector floating point unit. The small solid circles are nodes; arrows are drawn from source nodes to hyperedges and from hyperedges to target nodes. In this example, the three nodes that serve as inputs for more than one source tentacle are enlarged; the central grey node is in addition a join node since it is the target of more than one hyperedge.



This particular join can be used by the scheduler to take some pressure off the floating point unit. If in a larger context, sharing of one of the constants used by the integer instructions becomes possible, the affected branch becomes preferable over the other branch since this reduces register pressure.

The flavour of this kind of joins is very similar to the instruction sequence alternatives produced by superoptimisers [10, 6], but we also use joins for alternatives that produce *different* results that still meet, for example, appropriate precision requirements. This approach is justified by the relational semantics we present in Sect. 5.

Further discussion of examples and the speed-ups we achieved using our approach can be found in [2].

Reachability in code graphs is defined via the node successor relation, where n' is a successor of n if there is an edge for which n is a source node and n' a target node. We use this to define two basic node and edge properties:

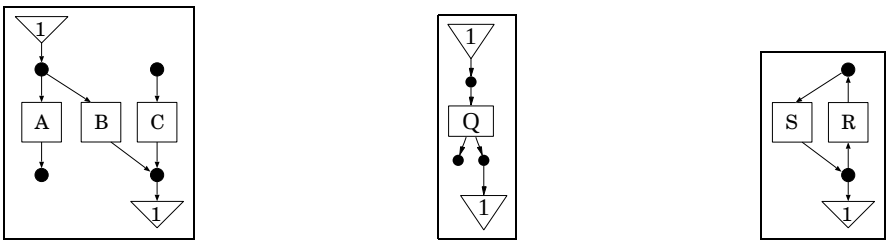
Definition 2.2 A node in a code graph is called *used* iff an output node is reachable from it, and *supported* iff it is either an input node or a target node of a supported edge.

An edge in a code graph is called *used* iff *at least one* of its target nodes is used, and *supported* iff *all* its source nodes are supported. □

The following code graphs provide some illustration of these concepts.

In the left graph, the A edge and its output node are unused, and the C edge and its input node are unsupported. In the middle graph, the left output node

of the Q edge is unused, but the Q edge itself is still used, since its right output node is an output node of the graph. In the right graph (which has one output and zero inputs), nothing is supported:



Building on top of these node and edge properties, we define a number of important graph properties:

Definition 2.3 A code graph is called:
- *acyclic* iff the node successor relation is acyclic,
- *join-free* iff each node occurs at most once in the concatenation of the target node lists of all edges with the input node list of the graph,
- *forward-garbage-free* iff all edges and non-output nodes are supported (no computations that cannot be performed because of lack of input),
- *backward-garbage-free* iff all *edges* are used (no computations for which no result is used),
- *garbage-free* iff it is both forward- and backward-garbage-free,
- *lean* iff it is garbage-free and join-free (and therefore acyclic),
- *coherent* iff all output nodes are supported,
- *solid* iff it is garbage-free and coherent,
- *executable* iff it is solid and lean. □

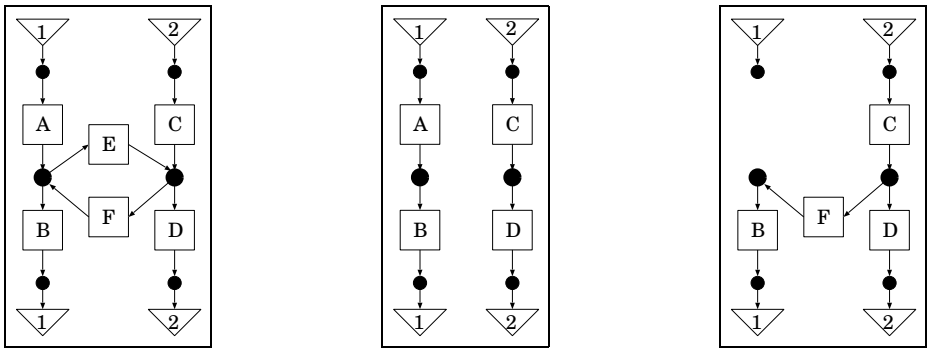
For example, the A edge above is backward-garbage (it will be collected in a backward direction) and the C edge is forward-garbage; the middle graph above is garbage-free, and furthermore lean and solid, and therefore executable, and the right graph has only forward-garbage edges and is not coherent.

The (forward-, resp. backward-) garbage-collected version of a code graph is obtained by iteratively deleting all nodes and edges that violate the respective condition — it is easy to see that the result is uniquely determined, always defined, and has the same input and output nodes as the original graph.

A first, simplified understanding of the use of code graphs in Coconut is the following:

- The code generator produces a coherent code graph G from a library of code graph fragments.
- G is garbage collected into a solid code graph S.
- The scheduler selects an executable subgraph E of S (or, more precisely, an executable "hyper-path" through S, see Sect. 5).
- The scheduler (already during selection) sequentialises the instructions in E in a way that maximises instruction-level parallelism in the target CPU.

Joins enable cycles, and since this may be particularly surprising in a data flow context, we discuss an (artificial) example here:



The graph on the left should be understood as describing a computation where the operations B and D require (at the hollow nodes) intermediate results that

- can be obtained from inputs (via A and C), and alternatively
- can be obtained from each other (via E resp. F).

The two graphs drawn beside it are both executable subgraphs that could be selected by the scheduler when processing the cyclic graph as input, and which of these will be more efficient may well depend on the context in which they are used.

3 Data-Flow Categories of Code Graphs

We now summarise the theory of our code graphs, which is essentially a reformulation of Ştefănescu's data-flow network algebra, in the language of category theory. In particular, we use the gs-monoidal categories proposed by Corradini and Gadducci for modelling acyclic term graphs [4].

The following definition serves mainly to introduce our notation:

Definition 3.1 A *category* $\mathbf{C}$ is a tuple $(\mathsf{Obj}, \mathsf{Mor}, \mathsf{src}, \mathsf{trg}, \mathbb{I}, \fatsemi)$ with the following constituents:

- Obj is a collection of *objects*.
- Mor is a collection of *arrows* or *morphisms*.
- src (resp. trg) maps each morphism to its source (resp. target) object. We write "$f : \mathcal{A} \to \mathcal{B}$" for "$f \in \mathsf{Mor} \wedge \mathsf{src}(f) = \mathcal{A} \wedge \mathsf{trg}(f) = \mathcal{B}$". The collection of all morphisms f of category $\mathbf{C}$ with $f : \mathcal{A} \to \mathcal{B}$ is denoted as $\mathsf{Mor}_{\mathbf{C}}[\mathcal{A}, \mathcal{B}]$ and also called a *homset*.
- "$\fatsemi$" is the binary *composition* operator, and composition of two morphisms $f : \mathcal{A} \to \mathcal{B}$ and $g : \mathcal{B}' \to \mathcal{C}$ is defined iff $\mathcal{B} = \mathcal{B}'$, and then $(f \fatsemi g) : \mathcal{A} \to \mathcal{C}$; composition is associative.
- $\mathbb{I}$ associates with every object $\mathcal{A}$ a morphism $\mathbb{I}_{\mathcal{A}}$ which is both a right and left unit for composition. □

The objects of the untyped code graph category over a set of edge labels ELab are natural numbers; in the typed case we would have sequences of types. A morphism from m to n is a code graph with m input nodes and n output nodes (more precisely, it is an isomorphism class of code graphs, since node and edge identities do not matter). Composition $F;G$ "glues" together the output nodes of F with the respective input nodes of G. The identity on n consists only of n input nodes which are also, in the same sequence, output nodes, and no edges.

A *primitive* code graph is a code graph that corresponds to a single operation, i.e., a code graph with a single edge where each node is the target of exactly one tentacle, and the target node sequence of the edge coincides with the output node sequence of the graph, and the source sequence with the input sequence.

Definition 3.2 A *symmetric strict monoidal category* $\mathbf{C} = (\mathbf{C}_0, \otimes, \mathbb{1}, \mathbb{X})$ consists of a category $\mathbf{C}_0$, a strictly associative monoidal bifunctor $\otimes$ with $\mathbb{1}$ as its strict unit, and a transformation $\mathbb{X}$ that associates with every two objects $\mathcal{A}$ and $\mathcal{B}$ an arrow $\mathbb{X}_{\mathcal{A},\mathcal{B}} : \mathcal{A} \otimes \mathcal{B} \to \mathcal{B} \otimes \mathcal{A}$ with:

$$(F \otimes G);\mathbb{X}_{\mathcal{C},\mathcal{D}} = \mathbb{X}_{\mathcal{A},\mathcal{B}};(G \otimes F) \ , \qquad \mathbb{X}_{\mathcal{A},\mathcal{B}};\mathbb{X}_{\mathcal{B},\mathcal{A}} = \mathbb{I}_{\mathcal{A}} \otimes \mathbb{I}_{\mathcal{B}} \ ,$$
$$\mathbb{X}_{\mathcal{A}\otimes\mathcal{B},\mathcal{C}} = (\mathbb{I}_{\mathcal{A}} \otimes \mathbb{X}_{\mathcal{B},\mathcal{C}});(\mathbb{X}_{\mathcal{A},\mathcal{C}} \otimes \mathbb{I}_{\mathcal{B}}) \ , \qquad \mathbb{X}_{\mathbb{1},\mathbb{1}} = \mathbb{I}_{\mathbb{1}} \ . \qquad \square$$

For code graphs, $\mathbb{1}$ is the number 0 and $\otimes$ on objects is addition. On morphisms, $\otimes$ forms the disjoint union of code graphs, concatenating the input and output node sequences. $\mathbb{X}_{m,n}$ differs from $\mathbb{I}_{m+n}$ only in the fact that the two parts of the output node sequence are swapped.

Definition 3.3 $\mathbf{C} = (\mathbf{C}_0, \otimes, \mathbb{1}, \mathbb{X}, !)$ is a *strict g-monoidal category* iff
– $(\mathbf{C}_0, \otimes, \mathbb{1}, \mathbb{X})$ is a symmetric strict monoidal category, and
– $!$ associates with every object $\mathcal{A}$ of $\mathbf{C}_0$ an arrow $!_{\mathcal{A}} : \mathcal{A} \to \mathbb{1}$,
such that $\mathbb{I}_{\mathbb{1}} = !_{\mathbb{1}}$, and *monoidality of termination* holds: $!_{\mathcal{A}\otimes\mathcal{B}} = !_{\mathcal{A}} \otimes !_{\mathcal{B}}$ $\square$

For code graphs, $!_n$ differs from $\mathbb{I}_n$ only in the fact that the output node sequence is empty. The "g" of "g-monoidal" stands for "garbage": all edges of code graph $G : m \to n$ are backward-garbage in $G;!_n$.

Note that $!_n$ itself is garbage free, coherent, and lean, and therefore solid and even executable.

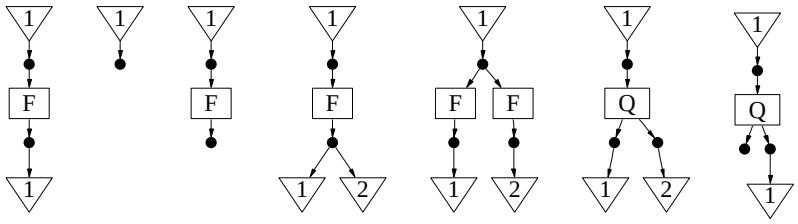
Definition 3.4 $\mathbf{C} = (\mathbf{C}_0, \otimes, \mathbb{1}, \mathbb{X}, \nabla)$ is a *strict s-monoidal category* $\mathbf{C}$ iff
– $(\mathbf{C}_0, \otimes, \mathbb{1}, \mathbb{X})$ is a symmetric strict monoidal category, and
– ∇ associates with every object $\mathcal{A}$ of $\mathbf{C}_0$ an arrow $\nabla_{\mathcal{A}} : \mathcal{A} \to \mathcal{A} \otimes \mathcal{A}$,
such that $\mathbb{I}_{\mathbb{1}} = \nabla_{\mathbb{1}}$, and the *coherence* axioms
– *associativity of duplication*: $\nabla_{\mathcal{A}};(\mathbb{I}_{\mathcal{A}} \otimes \nabla_{\mathcal{A}}) = \nabla_{\mathcal{A}};(\nabla_{\mathcal{A}} \otimes \mathbb{I}_{\mathcal{A}})$,
– *commutativity of duplication*: $\nabla_{\mathcal{A}};\mathbb{X}_{\mathcal{A},\mathcal{A}} = \nabla_{\mathcal{A}}$
and the *monoidality* axiom
– *monoidality of duplication*: $\nabla_{\mathcal{A}\otimes\mathcal{B}};(\mathbb{I}_{\mathcal{A}} \otimes \mathbb{X}_{\mathcal{B},\mathcal{A}} \otimes \mathbb{I}_{\mathcal{B}}) = \nabla_{\mathcal{A}} \otimes \nabla_{\mathcal{B}}$
are satisfied. $\square$

For code graphs, ∇_n differs from $\mathbb{I}_n$ only in the fact that the output node sequence is the concatenation of the input node sequence with itself. The "s" of "s-monoidal" stands for "sharing: every input of $\nabla_k{;}(F \otimes G)$ is shared by $F : k \to m$ and $G : k \to n$.

Definition 3.5 $\mathbf{C} = (\mathbf{C}_0, \otimes, \mathbb{1}, \mathbb{X}, \nabla, !)$ is a *strict gs-monoidal category* iff
- $(\mathbf{C}_0, \otimes, \mathbb{1}, \mathbb{X}, !)$ is a strict g-monoidal category, and
- $(\mathbf{C}_0, \otimes, \mathbb{1}, \mathbb{X}, \nabla)$ is a strict s-monoidal category,

such that the *coherence* axiom
- *right-inverse of duplication* holds: $\nabla_{\mathcal{A}}{;}(\mathbb{I}_{\mathcal{A}} \otimes !_{\mathcal{A}}) = \mathbb{I}_{\mathcal{A}}$ □

Code graphs (and term graphs) over a fixed edge label set form a gs-monoidal category, but not a *cartesian* category, where in addition ! and ∇ are *natural* transformations, i.e., for all $F : \mathcal{A} \to \mathcal{B}$ we have $F{;}!_{\mathcal{B}} = !_{\mathcal{A}}$ and $F{;}\nabla_{\mathcal{B}} = \nabla_{\mathcal{A}}{;}(F \otimes F)$. To see how these naturality conditions are violated, the first five code graphs in the following drawing can be obtained as, in this sequence, $F : 1 \to 1$, $!_1$, $F{;}!_1$, $F{;}\nabla_1$, and $\nabla_1{;}(F \otimes F)$:



It is easy to see that we obtain naturality of termination if we consider equivalence classes of code graphs up to backward-garbage collection. Therefore, we introduce a special "garbage-collecting" variant:

Definition 3.6 A *gc-s-monoidal category* is a gs-monoidal category with natural termination, i.e., with $G{;}!_{\mathcal{B}} = !_{\mathcal{B}}$ for all $G : \mathcal{A} \to \mathcal{B}$. □

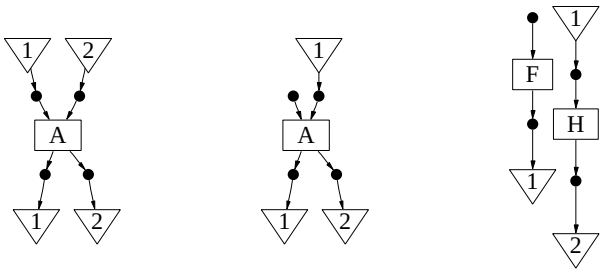
From the last two code graphs drawn above, namely $Q : 1 \to 2$ and $Q{;}(!_1 \otimes \mathbb{I}_1)$, we can also see why backward-garbage collection had to be defined so carefully: In the last graph, the Q edge is not backward-garbage since one of its results is still needed as output.

The code graph definition itself is completely symmetric with respect to inputs and outputs, so the duals of the termination and duplication are defined, too, and also satisfy all the corresponding laws, thus turning the category of code graphs over a set of primitives (with arities) into the free bi-gs-monoidal category over these primitives, or, equivalently, the free data-flow network algebra [14].

The dual to duplication is *join* $\Delta_{\mathcal{A}} : \mathcal{A} \otimes \mathcal{A} \to \mathcal{A}$, which, as a code graph, differs from $\mathbb{I}_{\mathcal{A}}$ only in the fact that the input node sequence is the concatenation of the output node sequence with itself.

The dual to termination is *co-termination* $\mathsf{i}_{\mathcal{A}} : \mathbb{1} \to \mathcal{A}$, which introduces forward-garbage and differs from $\mathbb{I}_{\mathcal{A}}$ only in the fact that the input node sequence is empty, so the corresponding code graph is not coherent.

Using primitives $F, H : 1 \to 1$ and $A : 2 \to 2$, we show in the following drawing the code graphs obtained as A alone, then $(\mathsf{i}_1 \otimes \mathbb{I}_1) ; A$, and finally $(\mathsf{i}_1 ; F) \otimes H$, which could also be obtained as $(\mathsf{i}_1 \otimes \mathbb{I}_1) ; (F \otimes H)$ because of functoriality of $\otimes$.



In the category of "code graphs up to forward garbage collection", co-termination is also a left-zero for composition, i.e., $\mathsf{i}_{\mathcal{A}} ; F = \mathsf{i}_{\mathcal{B}}$ for every $F : \mathcal{A} \to \mathcal{B}$, and also satisfies for each primitive $P : \mathcal{A} \to \mathcal{B}$ and each decomposition $\mathcal{A} = \mathcal{A}_1 \otimes \mathcal{A}_2 \otimes \mathcal{A}_3$ the following equation which corresponds in more detail to forward garbage collection:

$$(\mathbb{I}_{\mathcal{A}_1} \otimes \mathsf{i}_{\mathcal{A}_2} \otimes \mathbb{I}_{\mathcal{A}_3}) ; P = \mathsf{i}_{\mathcal{B}}$$

However, not even there we have $F \otimes \mathsf{i}_{\mathcal{C}} = \mathsf{i}_{\mathcal{B} \otimes \mathcal{C}}$ in general for $F : \mathcal{A} \to \mathcal{B}$.

All important kinds of code graphs as morphisms form at least gs-monoidal categories (co-s-monoidal categories also have joins and the corresponding laws, and co-g-monoidal categories correspondingly have co-termination):

Definition 3.7 With operations as defined above, natural numbers as objects, and given primitives (i.e., edge labels with input and output arities) we define the following categories:
- CG with code graphs as morphisms,
- CCG with coherent code graphs as morphisms.

Replacing direct code graph composition with composition that "performs automatic garbage collection", we further define:
- LCG with lean code graphs as morphisms,
- SCG with solid code graphs as morphisms,
- ECG with executable code graphs as morphisms. □

Proposition 3.8 The categories in Def. 3.7 are well defined, and, with primitive code graphs as generators, we have:
- CG is the free bi-gs-monoidal category;
- CCG is the free gs-monoidal and co-s-monoidal category;
- LCG is the free gc-s-monoidal and co-g-monoidal category;
- SCG is the free gc-s-monoidal and co-s-monoidal category;
- ECG is the free gc-s-monoidal category. □

4 Control Flow Aspects of Code Graphs

As already mentioned in Sect. 2, the task of the scheduler is to find (efficient) executable "hyper-paths" through a solid code graph.

If both graphs and edges were restricted to be one-input and one-output, then branching could occur only for the purpose of later joins, and a code graph would become a finite automaton, where the input node is the start state, the output node is the (only) accepting state, and edges are labelled with machine instructions. For such an automaton, we obtain its *operational semantics* in the *free Kleene algebra* of sets of instruction execution sequences.

Since each machine instruction induces a state transition relation, and this in turn induces another state transition relation for each set of instruction execution sequences, we obtain the *denotational semantics* of such an automaton in the *Kleene algebra* of relations.

To prepare the generalisation to arbitrary code graphs, we note that a more graph-theoretic view of the operational semantics is to consider it as the set of all paths from input to output, or, equivalently (and easier to generalise), as the set of all *line graphs* for which there is an input- and output-preserving homomorphism into the original code graph.

In comparison with finite automata, the additional code graph features are term graph features, namely

- n-ary operations enabled by parallel composition $\otimes$,
- multiple use of results, enabled by duplication ∇, and
- unused (additional) results, enabled by termination $!$.

Therefore, we need to enrich the Kleene algebra semantics with gs-monoidal features; we only need the technically simple complete variant of Kleene categories (see also [8]):

Definition 4.1 A *locally ordered category* is a category $\mathbf{C}$ such that
- for each two objects $\mathcal{A}$ and $\mathcal{B}$, the relation $\sqsubseteq_{\mathcal{A},\mathcal{B}}$ is a partial order on the homset $\mathsf{Mor}_{\mathbf{C}}[\mathcal{A},\mathcal{B}]$ (the indices will usually be omitted), and
- composition is monotonic with respect to $\sqsubseteq$ in both arguments.

A *complete Kleene category* is a locally ordered category where each homset is a complete upper semilattice and composition distributes over arbitrary joins from both sides. □

This implies the existence of zero morphisms $\perp\!\!\!\perp$, binary union, and Kleene star, all obeying the usual laws for typed Kleene algebras [9].

Now the denotational semantics of code graphs can use the gs-monoidal Kleene category of relations. For the operational semantics, we use the standard construction of set-based Kleene categories:

It is well-known that for any category $\mathbf{C} = (\mathsf{Obj}_{\mathbf{C}}, \mathsf{Mor}_{\mathbf{C}}, \mathsf{src}, \mathsf{trg}, \mathbb{I}_{\mathbf{C}}, ;_{\mathbf{C}})$, a complete Kleene category $\mathbf{C}^{\mathbb{P}}$ can be obtained by defining its components as follows:

– The objects are the same: $\mathsf{Obj}_{\mathbf{C}^{\mathbb{P}}} = \mathsf{Obj}_{\mathbf{C}}$
– Morphisms are subsets of the corresponding $\mathbf{C}$-homsets:

$$\mathsf{Mor}_{\mathbf{C}^{\mathbb{P}}}[\mathcal{A}, \mathcal{B}] = \mathbb{P}\ \mathsf{Mor}_{\mathbf{C}}[\mathcal{A}, \mathcal{B}]$$

– Identities are singletons: $\mathbb{I}_{\mathbf{C}^{\mathbb{P}},\mathcal{A}} = \{\mathbb{I}_{\mathbf{C},\mathcal{A}}\}$
– Composition is set composition: $F \fatsemi_{\mathbf{C}^{\mathbb{P}}} G = \{f \fatsemi_{\mathbf{C}} g \mid f \in F \wedge g \in G\}$
– The ordering is set inclusion: $F \sqsubseteq G \Leftrightarrow F \subseteq G$. This ordering is complete, and composition distributes over arbitrary joins, so we have:
  • Least elements: $\perp\!\!\!\perp_{\mathcal{A},\mathcal{B}} = \{\}$
  • Binary joins: $F \sqcup G = F \cup G$
  • If $F : \mathcal{A} \to \mathcal{A}$, then Kleene star: $F^* = \bigcup\{F^n \mid n \in \mathbb{N}\}$, with the understanding that $\mathbb{N} = \{0, 1, 2, \ldots\}$, and $F^0 = \mathbb{I}_{\mathcal{A}}$ and $F^{n+1} = F \fatsemi F^n$.

This construction preserves gs-monoidality:

Theorem 4.2 If $\mathbf{C} = (\mathbf{C}_0, \otimes, \mathbb{1}, \mathbb{X}, \nabla, !)$ is a gs-monoidal category, then we obtain a gs-monoidal category, again, by extending $\mathbf{C}^{\mathbb{P}}$ with the following constants:
– $\otimes$ and $\mathbb{1}$ on objects are the same as in $\mathbf{C}$.
– monoidal composition is defined as monoidal set composition:

$$F \otimes_{\mathbf{C}^{\mathbb{P}}} G = \{f \otimes g \mid f \in F \wedge g \in G\}$$

– the constants are singleton sets:
  • $\mathbb{X}_{\mathbf{C}^{\mathbb{P}},\mathcal{A},\mathcal{B}} = \{\mathbb{X}_{\mathcal{A},\mathcal{B}}\}$
  • $!_{\mathbf{C}^{\mathbb{P}},\mathcal{A}} = \{!_{\mathcal{A}}\}$
  • $\nabla_{\mathbf{C}^{\mathbb{P}},\mathcal{A}} = \{\nabla_{\mathcal{A}}\}$

Proof: This gs-monoidal category is well defined since all constants are defined as singleton sets, and in each axiom of gs-monoidal categories, all variables occur exactly once on *both* sides of the equality in such a way that the resulting sets are always isomorphic via axiom instances on the elements. □

5 Code Graph Semantics and Scheduling

We now are in a position to provide the details of the semantical aspects of the use of code graphs in Coconut as sketched in Sect. 2.

The principle we use is that of *functorial semantics* [5]: Since all our gs-monoidal code graph categories are, according to Proposition 3.8, freely generated from the primitives modulo some additional constants and/or laws, any gs-monoidal category $\mathbf{C}$ providing these constants and laws immediately provides a semantics $[\![G]\!]^{\mathbf{C}}$ for each code graph G from the respective code graph category via the unique gs-monoidal functor from the free (i.e., initial) category into the chosen semantical category.

– We start with a (total) relational specification R.

– The code generator produces a coherent code graph G from a library of code graph fragments — G is a morphism of CCG. The denotational semantics of CCG is considered in the gs-monoidal and co-s-monoidal category of *total relations*, where all primitives are interpreted as *total functions* (we do not consider halting or interrupting machine instructions).
  The task of the code generator therefore is to ensure that $[\![G]\!]^{TotRel} \sqsubseteq R$.
– G is garbage collected into a solid code graph S, a morphism of SCG, still interpreted as the same total relation $[\![S]\!]^{TotRel} = [\![G]\!]^{TotRel}$.
– The scheduler selects an executable code graph E, i.e., a morphism of ECG, from the set of executable code graphs that form the functorial semantics $[\![S]\!]^{\mathsf{ECG}^{\mathbb{P}}}$ of S in the gs-monoidal Kleene category $\mathsf{ECG}^{\mathbb{P}}$.
  This selection is of course not arbitrary, but attempts to select a code graph that is minimal to some resource consumption metric, normally execution time, or, more precisely, total throughput through the resulting program, occasionally influenced by register consumption. On a pipelined architecture, these metrics are not fully compositional, so the gs-monoidal Kleene category structure is only of limited use for designing appropriate optimisation strategies for the scheduler.

  Executable code graphs are interpreted in the cartesian category of total functions. Since $\{E\} \sqsubseteq [\![S]\!]^{\mathsf{ECG}^{\mathbb{P}}}$ in $\mathsf{ECG}^{\mathbb{P}}$, we also have $[\![E]\!]^{TotRel} \sqsubseteq [\![S]\!]^{TotRel}$ in the category of total relations, which establishes that E with its functional semantics $[\![E]\!]^{Set} = [\![E]\!]^{TotRel}$ satisfies its relational specification $[\![S]\!]^{TotRel}$ and therefore R.
– By construction, there is a code graph homomorphism from the executable (and therefore join-free) code graph E to S, so E can also be considered as a generalised path ("hyper-path") through S — the scheduler is therefore implemented as a kind of shortest path search.

6 Conclusion and Outlook

The code graphs introduced in Sect. 2 serve as concrete syntax for computation fragments at the assembly level in the special-purpose compiler suite of the Coconut project. We have shown that through the device of functorial semantics, these code graphs, essentially considered as data-flow graphs, can be equipped with a relational denotational semantics, which is used for reasoning about their correctness.

The joins in these data-flow graphs have been assigned a novel interpretation which amounts to giving control-flow semantics to this aspect of data-flow graphs — this is realised by defining a functorial operational semantics in a Kleene category of "hyper-paths".

While joins give us obviously the opportunity to integrate a superoptimiser into the code generator similar to [6], the fact that we use a relational semantics also allows us to be more flexible and make use of the fact that, for example, many results in image processing only need to be correct up to a relatively low precision, so mathematically radically different algorithms can be explored.

The coherent semantical treatment of code graphs as presented in Sect. 5 has already proven beneficial in guiding the design of the scheduler for computation fragments without branches and loops.

The most important next step is to integrate this with proper control flow structure, in particular for wrapping loop structures around computation fragments serving as loop bodies, and for the separation of loop bodies into several stages for the purpose of modulo scheduling [1] which is another useful optimisation trick in the context of heavily pipelined architecture.

We would like to thank the anonymous referees for their useful comments.

References

[1] V. H. Allan, R. B. Jones, R. M. Lee, S. J. Allan. *Software pipelining.* ACM Comput. Surv. **27**(3) 367–432, 1995.

[2] C. K. Anand, J. Carette, W. Kahl, C. Gibbard, R. Lortie. *Declarative Assembler.* SQRL Report 20, Software Quality Research Laboratory, McMaster University, 2004. available from http://sqrl.mcmaster.ca/sqrl_reports.html.

[3] M. Coccia, F. Gadducci, A. Corradini. *GS-Λ Theories: A Syntax for Higher-Order Graphs.* Electronic Notes in Computer Science **69** 18, 2002.

[4] A. Corradini, F. Gadducci. *An Algebraic Presentation of Term Graphs, via GS-Monoidal Categories.* Applied Categorical Structures **7**(4) 299–331, 1999.

[5] A. Corradini, F. Gadducci. *Functorial Semantics for Multi-Algebras and Partial Algebras, with Applications to Syntax.* Theoretical Computer Science **286**(2) 293–322, 2002.

[6] T. Granlund, R. Kenner. *Eliminating Branches Using a Superoptimizer and the GNU C Compiler.* In: Programming Language Design and Implementation, PLDI '92, pp. 341–352. acm, 1992.

[7] B. Hoffmann, D. Plump. *Jungle Evaluation for Efficient Term Rewriting.* In J. Gabrowski, P. Lescanne, W. Wechler, eds., Algebraic and Logic Programming, ALP '88, Mathematical Research **49**, pp. 191–203. Akademie-Verlag, 1988.

[8] W. Kahl. *Refactoring Heterogeneous Relation Algebras around Ordered Categories and Converse.* J. Relational Methods in Comp. Sci. **1** 277–313, 2004.

[9] D. Kozen. *Typed Kleene Algebra.* Technical Report 98-1669, Computer Science Department, Cornell University, 1998.

[10] H. Massalin. *Superoptimizer: A Look at the Smallest Program.* In: ASPLOS-II: Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems, pp. 122–126, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[11] D. Plump. *Term Graph Rewriting.* In H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg, eds., Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools, Chapt. 1, pp. 3–61. World Scientific, Singapore, 1999.

[12] M. Sleep, M. Plasmeijer, M. van Eekelen, eds. *Term Graph Rewriting: Theory and Practice.* Wiley, 1993.

[13] Gheorghe Ştefănescu. *Algebra of Flownomials — Part 1: Binary Flownomials; Basic Theory.* Technical Report TUM-I9437, Technische Universität München, Institut für Informatik, 1994.

[14] Gheorghe Ştefănescu. *Network Algebra.* Springer, London, 2000.