An Algebraic Approach to Parameterised Loop Decomposition

An Algebraic Approach to Parameterised Loop Decomposition

By
Shiqi Cao B.Sc. (Hons)

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree
Master of Science

McMaster University

MASTER OF SCIENCE(2008) McMaster University
COMPUTING AND SOFTWARE Hamilton, Ontario

TITLE: An Algebraic Approach to Parameterised Loop Decomposition

AUTHOR: Shiqi Cao B.Sc. (Hons)(McMaster University)

SUPERVISOR: Dr. Wolfram Kahl

NUMBER OF PAGES: ix, 70

Abstract

Loop scheduling is to explore more possible parallelism by re-organizing the loop body without changing its semantics; it results in more efficient utilization of the underlying hardware. Recently, research has been shifting from well-studied instruction level parallelism to thread level parallelism (TLP) in order to follow the trends of CPU design; parts of the COCONUT project are moving in this direction as well. Loops are usually represented in graph-like structures, which, without algebraic properties, can make formal verification very difficult.

In this thesis, a new representation of a loop, called an extensible loop specification, is proposed, based on the code graph and loop specification concepts already used in the COCONUT code generator. Extensible loop specifications are intended to be used by TLP loop scheduling algorithms; their algebraic properties derive from those of loop specifications and code graphs.

During the process of discovering a new loop representation, we use a relational model to verify some transformations of control flow graphs where transitions are labeled with code graphs.

Acknowledgments

First, I would like to sincerely thank my supervisor, Dr. Wolfram Kahl, for his guidance, supervision and support over past several years.

I would like to thank Scott West for his comments on my thesis.

I would also like to thank my aunt and her family for making me feel at home during my studies in Canada. Finally, I would like to thank my parents for everything they have done for me.

Contents

List of Figures

Chapter 1

Introduction

Correctness and efficiency are two different aspects of software development; some applications like image and signal processing for medical purposes are highly dependent on both. Technically, the two aspects do not cooperate well; correctness, usually, is easy to deal with at a higher level in development. Improving efficiency usually does not happen until several intermediate transformations before implementation. During these several transformations, correctness information becomes difficult to trace or may even be lost; ideally, correctness information should be accessible, which would allow efficiency to be improved while preserving correctness.

The COCONUT (COde CONstructing User Tool) project aims to produce a software development framework, created especially for safety-critical, high-performance scientific software. COCONUT emphasizes two areas, formal capture of the entire design, and formal transferring between development stages; we believe that with these two goals correctness and efficiency will be simultaneously respected in every development stage. Capturing the entire design requires a rich internal representation, and formal transformation requires the representation to be rigorous; a good example is term graph with choices introduced in [KAC06a]. This thesis mainly formalizes and enhances some representations used in COCONUT, and also proposes a new formal representation for future development. The contribution of this thesis includes two parts,

- A new structure introduced in Chapter 4_{19}, the extensible loop specification, is defined with algebraic properties. Extensible loop specification is base on loop specification [Tha06]; to make the foundation solid, in Chapter 3_7 we present loop specification with a formal semantics.
- During exploring extensible loop specification, we also verify some graph

transformations used in [AK08a], it is presented in Chapter 5_{37} with introduction of relation-weighted control graphs(RWCFG) [SS93; SHW97].

The two parts of this thesis are not tightly related; only in Section 3.4_{16} we show how some loop specifications can be embedded into RWCFGs directly. Chapter 2_3 summarizes code graphs [KAC06b] with algebraic properties; which is the foundation of loop specification(Chapter 3_7). Chapter A_{49} provides a basis in relation algebra which is required in Chapter 5_{37}.

Chapter 2

Code Graph and Semantics

This chapter serves as an entry point to Chapter 3_7, it includes a summery of code graph and some algebraic properties. Code graph is first introduced in [KAC06b]; if you have read [KAC06b] or familiar with the topic, feel free to skip this chapter.

A code graph [KAC06b] is a directed hypergraph with a sequence of input and output interface; the interface consists of two node sequences; each edge is a function symbol, and nodes are labeled with type information.

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.* □

An acyclic code graph is used to represent a term graph, which is conventionally represented by a non-hyper graph where nodes represent function symbols and edges connect function calls with their arguments; whereas the role of nodes and edges are switched in the code graph. A code graph is more expressive than a non-hyper graph. In non-hyper graphs, there is a trade off between exactly only one output of each function symbol and variable sharing;

if exactly only one output is enforced, every output tentacle from a node shares the same variable, then a pre-defined "projector" nodes can be used to fake multiple outputs. If each output tentacle from a node represents an output, then variable sharing must be faked by a pre-defined node. The code graph structure is rich enough to naturally express both; moreover, semantics of a code graph node with more than one input tentacle in [KAC06b] is defined to mean that the same result can be obtained in different ways.

There are many properties of code graphs defined in [KAC06b], we select a minimum set which are used in later chapters.

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.* □

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.* □

The theory of code graphs is formulated in the language of category theory, in particular gs-monoidal categories over a set of primitive code graphs. [KAC06b] contains all definitions, explanations and examples, we only present a brief summary for later chapters. The next four definitions are presented in hierarchy of dependency; Def. 2.4_4 introduces *symmetric strict monoidal category*, which serves as a base; Def. 2.5_4 and Def. 2.6_5 are defined based on Def. 2.4_4; then Def. 2.7_5 combines Def. 2.5_4 and Def. 2.6_5 together.

Definition 2.4 *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:*

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

Definition 2.5 $\mathbf{C} = (\mathbf{C}_0, \otimes, \mathbb{1}, \mathbb{X}, \nabla)$ *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. □

Definition 2.6 $\mathbf{C} = (\mathbf{C}_0, \otimes, \mathbb{1}, \mathbb{X}, !)$ *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:* $!_{\mathcal{A}\otimes\mathcal{B}} = !_{\mathcal{A}} \otimes !_{\mathcal{B}}$ □

Definition 2.7 $\mathbf{C} = (\mathbf{C}_0, \otimes, \mathbb{1}, \mathbb{X}, \nabla, !)$ *is a* strict gs-monoidal category *iff*

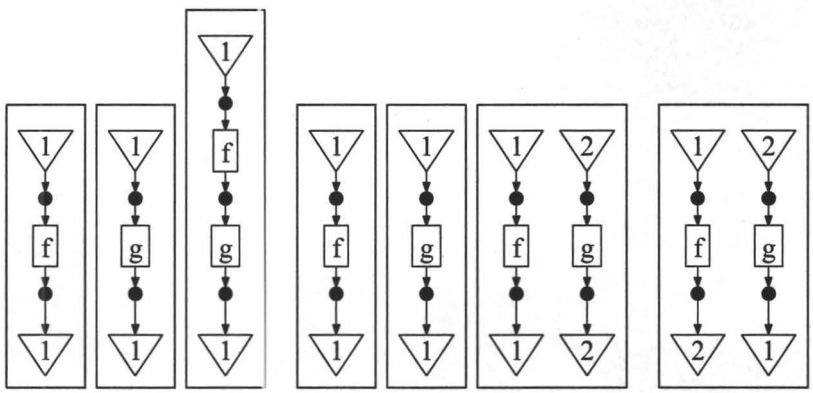
- $(\mathbf{C}_0, \otimes, \mathbb{1}, \mathbb{X}, !)$ *is a strict g-monoidal category, and*
- $(\mathbf{C}_0, \otimes, \mathbb{1}, \mathbb{X}, \nabla)$ *is a strict s-monoidal category,*

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 can be considered as arrows between objects which are sequences of node labels.

$\otimes$ is a bifunctor, for code graph, in parallel it composes two code graphs and input and output node sequences are concatenated. Object $\mathbb{1}$ is the left and right unit of $\otimes$, the type of empty sequence corresponds to $\mathbb{1}$ in code graph. ∇ duplicates input node sequence at output, it makes sharing inputs possible. $\mathbb{X}_{m,n}$ differs from $\mathbb{I}_{m\otimes n}$ only in the fact that two parts of output are swapped. $!$ terminates input by outputting an empty sequence. i_n is dual to $!_n$ in the face that input is empty but the output is typed to n. The following figures demonstrate the operators, f and g are two code graphs with only one edge, one input, one output.



(a) $f;g$ (b) $f \otimes g$ (c) $(f \otimes g);\mathbb{X}$

(d) $f;\nabla$ (e) $f;!$ (f) $\mathrm{i};f$

Chapter 3

Loop Specification and Semantics

Loop scheduling is an active topic in static program optimization in the field of compiler design. However there is no uniform presentation of loops; some are more like term graphs and some are more in the flavour of control flow graph. The soundness of transformations of these graphs is not discussed formally due to the lack of algebraic properties. A loop specification was introduced in [Tha06] for the COCONUT project. It is defined based on code graphs. Loop carried dependencies are captured at interface of code graphs; each loop carried dependency is assigned an integer as loop distance.

The meaning of loop specification is carefully explained in [Tha06]. In Section 3.2_9 we take a further step to interpret it with an operational semantics, which translates a loop specification into a set of code graphs. A formal semantics is intended to be used as a verification framework for loop transformations. Moreover, it is desirable to base development (Chapter 4_{19}) on a well defined and rigorous foundation.

In Section 3.4_{16}, we show an implementation of simplified loop specifications in RWCFG, the simplification algorithm is presented in Section 3.3_{15}. Wrapped by RWCFG vividly shows how loop-carried dependencies are connected.

3.1 Loop Specification

Loop specification is an abstract presentation of loop, it abstracts one aspect of loop, it omits how values are passed from different iterations, it only specifies which iteration a value is from.

A Loop Specification is a pair $(G, \mathbf{d})$, G is a code graph with signature $G : K \otimes F \to C \otimes F$. K is the type of constants, it is an initial input to the loop body and its value does not change in every iteration. C is the type of control information, its values are intended to be used to decide termination of the loop. F is the type of loop carried values. $\mathbf{d}$ assigns an integer to each element in F. The meaning of $\mathbf{d}(x) = d$ of the element x in F is that this input to current iteration was generated in d iteration(s) ago. In [Tha06], d iteration(s) ago is denoted by $-d$, but backwards dependency are more common; our way can avoid double negation when reading it. Besides this opposite treatment of d, this thesis also uses a different arrangement of the code graph interface from the original loop specification [Tha06], which used $X : F \otimes K \to F \otimes C$.

For example, let $d = 2$ of an input, then the value it needs was generated two iterations ago. As an example, Figure 3.1_s arises in an implementation

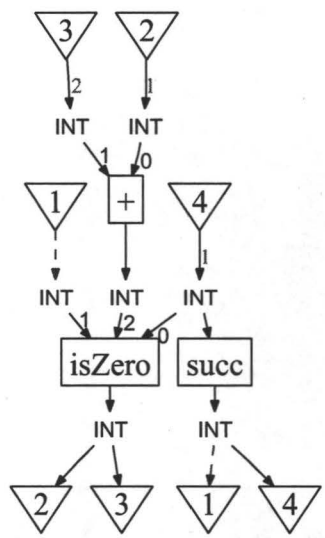


Figure 3.1: Fibonacci

of generating Fibonacci numbers. The sequences of inputs and outputs are indicated by arrows from, respectively to, numbered triangles. Type K is constituted by red dashed arrow and C by blue dashed arrow. The black arrows from triangles denote type F, the integer on each arrow is loop distance. Named rectangles denote hyperedges, if there are more than one input or output to or, respectively from, a hyperedge, then arrows with integers reflect their positions in the sequences.

Input 1 is a constant of type K. Output 1 returns control information. Inputs and outputs 2, 3, 4 constitute a sequence of type F. Input 4 is a

counter, in each iteration the hyperedge "succ" increases the counter by 1 and the result is used both as a loop carried value and control information. "isZero" returns its second input if the first input is equal to 0 otherwise it returns the third input. In this example $\mathbf{d} = \{2 \mapsto 1, 3 \mapsto 2, 4 \mapsto 1\}$, Input 3 consumes the output from 2 iterations ago.

Let d be assigned to a loop carried input, $d < 0$ means the input depends on the value generated after d iterations in the future. An input with $d = 0$ means an internal direct dependency being exposed intentionally, it can be hidden in the loop body completely. In this chapter we restrict d to be greater than 0. The main reason to drop $d \leqslant 0$ is that forward dependency is used with assumed implementation, like the staged-pipeline method, in mind, interpreting forward dependency with a specific implementation is not general enough; also loop specifications with forward dependency usually are transformed from non-forward dependency loop specification, in our opinion the emphasis should be on how to reason the transformation other than how to interpret forward dependency. In the rest of this chapter, we present a formal semantics for loop specification, then we show it can be converted to $d = 1$ for any input with $d > 1$.

3.2 Semantics

Operational semantics usually translates source language to well defined mathematical objects; therefore code graphs are an ideal codomain; this makes translation relatively simple because a loop specification is defined based on code graphs. The technique of translation is similar to unfolding a loop; given a number of iterations j, then j instances of the loop body are serially combined with carefully connecting loop carried dependency interfaces. Therefore for each non-negative integer there is a corresponding code graph, a loop specification is translated to an infinite set of code graphs. The rest of this section explains details of the translation with examples.

An interface shuffling is an auxiliary definition. When interpreting a loop specification, loop-carried dependencies with $d = 1$ and $d > 1$ are treated differently. $S_{\mathbf{G}}$ is a "function" that takes a sequence of loop carried dependency interface and returns two sequences, one for $d = 1$ and the other for $d > 1$. The proof of existence of $S_{\mathbf{G}}^{\sim}$ is also included in the definition since it simply applies the definition of permutations from [CG99].

Definition 3.1 *Let* $\mathbf{G}$ *be a loop specification,* $\mathbf{G} = (G, \boldsymbol{d})$ *and* $G : K \otimes F \to C \otimes F$. *A shuffle* $S_{\mathbf{G}}$ *is a permutation[CG99] of* F *according to* $\boldsymbol{d}$. $S_{\mathbf{G}}$ *shuffles* F *into two parts* $A \otimes B$, $\boldsymbol{d}(A) = 1$ *and* $\boldsymbol{d}(B) > 1$. $S_{\mathbf{G}}$ *can be implemented only*

by $\mathbb{X}$ *and* $\mathbb{I}$. $S_{\mathbf{G}}^{\sim}$ *is a permutation as well, then there is an implementation of it according to [CG99].* □

Definition 3.2 *Let* $\mathbf{G}$ *be a loop specification,* $\mathbf{G} = (G, \boldsymbol{d})$. $G : K \otimes F \to C \otimes F$ *is the signature of code graph* G *and* $S_{\mathbf{G}} : F \to A \otimes B$. $[\![\mathbf{G}]\!]_j$ *is a code graph of* j *iterations.* $\mathbf{G}' = (G', \boldsymbol{d}')$ *is another loop specification defined based on* $\mathbf{G}$.

$G' : K \otimes A \otimes B \otimes B' \to K \otimes A \otimes B \otimes B'$, *where* $B' = B$

$G' = (\nabla_K \otimes \mathbb{I}_{A\otimes B\otimes B'})\fatsemi(\mathbb{I}_K \otimes ((\mathbb{I}_K \otimes S_{\mathbf{G}}^{\sim})\fatsemi G\fatsemi(!_C \otimes S_{\mathbf{G}})) \otimes \mathbb{I}_{B'})\fatsemi(\mathbb{I}_{K\otimes A} \otimes \mathbb{X}_{B,B'})$

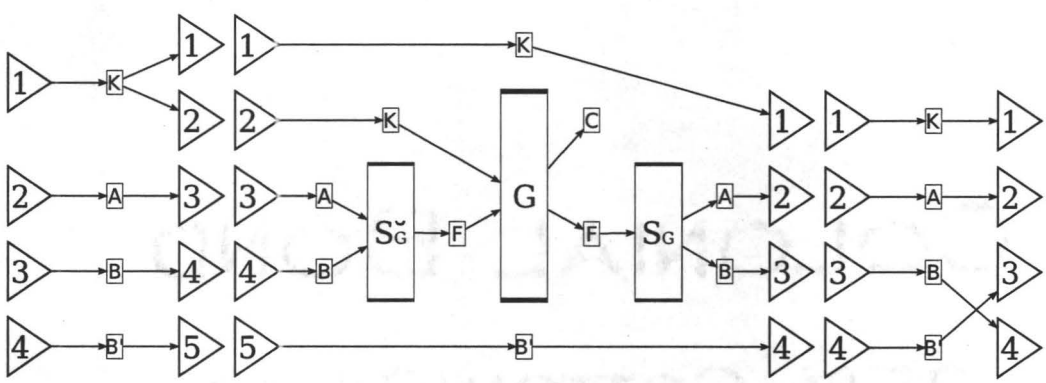
$\boldsymbol{d}'(B') = \boldsymbol{d}(B) - 1$ *and* $\boldsymbol{d}'(A \otimes B) = 1$.

Then $[\![\mathbf{G}]\!]_j$ *is defined as following, let* $W(X) = (\mathbb{I}_K \otimes S_{\mathbf{G}})\fatsemi X\fatsemi(\mathbb{I}_K \otimes S_{\mathbf{G}}^{\sim})\fatsemi G$

$$[\![\mathbf{G}]\!]_j = \begin{cases} W(\mathbb{I}_{K\otimes A} \otimes !_B \otimes i_B), \text{if } j = 1 \\ W((\mathbb{I}_{K\otimes A} \otimes i_B \otimes \mathbb{I}_{B'})\fatsemi[\![\mathbf{G}']\!]_{j-1}\fatsemi(\mathbb{I}_{K\otimes A} \otimes \mathbb{I}_B \otimes !_{B'})), \text{if } j > 1 \end{cases}$$

□

G' is an auxiliary "loop specification" defined based on the shuffled G. G' reduces loop distances to 1 for original inputs with loop distance greater than 1; therefore $\mathbf{d}'$ assigns all inputs typed by $A \otimes B$ to 1. The extended interface B' is equal to B and all inputs in B' are 1 less than the counterpart in B. Syntactically G' is a code graph since $K = K$, $F = A \otimes B \otimes B'$, $C = K$. However, the output C is not used as control information. In the definition of G', $!_C$ terminates C from G and wiring K to the position of C.



(a) $\nabla_K \otimes \mathbb{I}_{A\otimes B\otimes B'}$ (b) $\mathbb{I}_K \otimes ((\mathbb{I}_K \otimes S_{\mathbf{G}}^{\sim})\fatsemi G\fatsemi(!_C \otimes S_{\mathbf{G}})) \otimes \mathbb{I}_{B'}$ (c) $\mathbb{I}_{K\otimes A} \otimes \mathbb{X}_{B,B'}$

Figure 3.2: Three components of G'

Figure 3.2_{10} depicts the implementation of G', it is represented as a serial composition of three components. The first component, Figure $3.2(a)_{10}$, just duplicates constant inputs. Figure $3.2(a)_{10}$ actually depicts $\nabla_K \otimes \mathbb{I}_A \otimes \mathbb{I}_B \otimes \mathbb{I}_{B'}$, which is equivalent to $\nabla_K \otimes \mathbb{I}_{A \otimes B \otimes B'}$. Although the former is wordily but its diagram is more readable than squeezing everything to node type. The second component, Figure $3.2(b)_{10}$, terminates the output C from G and bypasses B' and K to the next component. G is surrounded by $S_{\widetilde{G}}$ and S_G in order to make some distance information available through new interface. The third component, Figure $3.2(c)_{10}$, swaps B and B' generated from previous component.

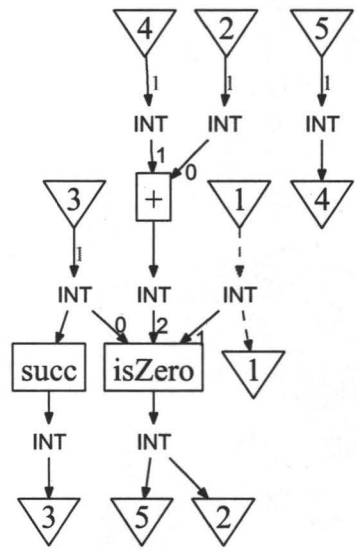
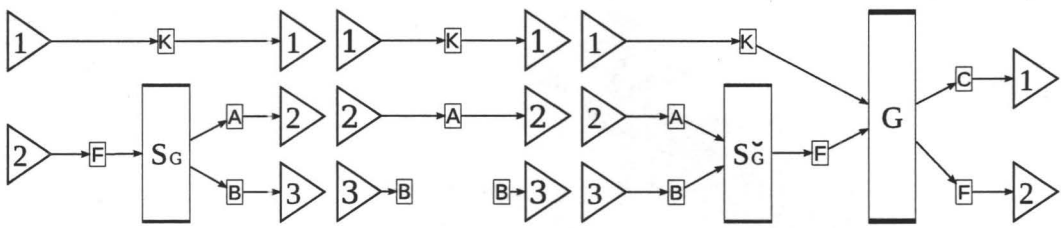


Figure 3.3: Fibonacci′

Figure 3.3_{11} is G', where G is loop specification generating Fibonacci numbers Figure 3.1_8. Output 1 was after *succ*; now it bypasses the constant, input 1. The original input interface was $K \otimes F$, K is input 1, F is 2 and 3 and 4. Following the Def. 3.1_9, A is constituted by input, referring to Figure 3.1_8, 2 and 4; B by input 3, since loop distance of input 3 is 2. Therefore input interface of Fibonacci′ swaps 3 and 4 from original G, because A contains all inputs with $d = 1$. Moreover the new input 5 is B', loop distance of it is $2 - 1$. The output interface bypasses input 1, terminates one of duplicated output of *succ* that was used as C. Swapping B and B' results input 5 is bypassed as output 4.

$[\![\mathbf{G}]\!]_j$ is defined inductively on, j, number of iterations. The definition splits into two cases, base case $j = 1$ and inductive part $j > 1$. Both terms

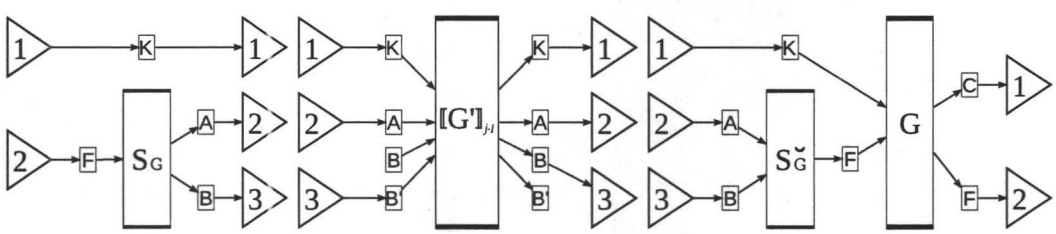
are wrapped by $(\mathbb{I}_K \otimes S_{\mathbf{G}})$;...;$(\mathbb{I}_K \otimes S^{\sim}_{\mathbf{G}})$;$G$, Figure 3.4(a)$_{12}$ and Figure 3.4(c)$_{12}$ respectively. The opening part, $\mathbb{I}_K \otimes S_{\mathbf{G}}$, ensures $[\![\mathbf{G}]\!]_j$ is typed by the same signature as G; this is convenient since the signature of $[\![\mathbf{G}']\!]_{j-1}$ is known when implementing $[\![\mathbf{G}]\!]_j$, otherwise the type information must be returned by $[\![\mathbf{G}']\!]_{j-1}$ somehow. As mentioned previously, the closing part, $\mathbb{I}_K \otimes S^{\sim}_{\mathbf{G}}$, exposes some distance information by S_G.



(a) $\mathbb{I}_K \otimes S_{\mathbf{G}}$ (b) $\mathbb{I}_{K\otimes A} \otimes !_B \otimes \mathsf{i}_B$ (c) $(\mathbb{I}_K \otimes S^{\sim}_{\mathbf{G}})$;$G$

Figure 3.4: $[\![\mathbf{G}]\!]_1$

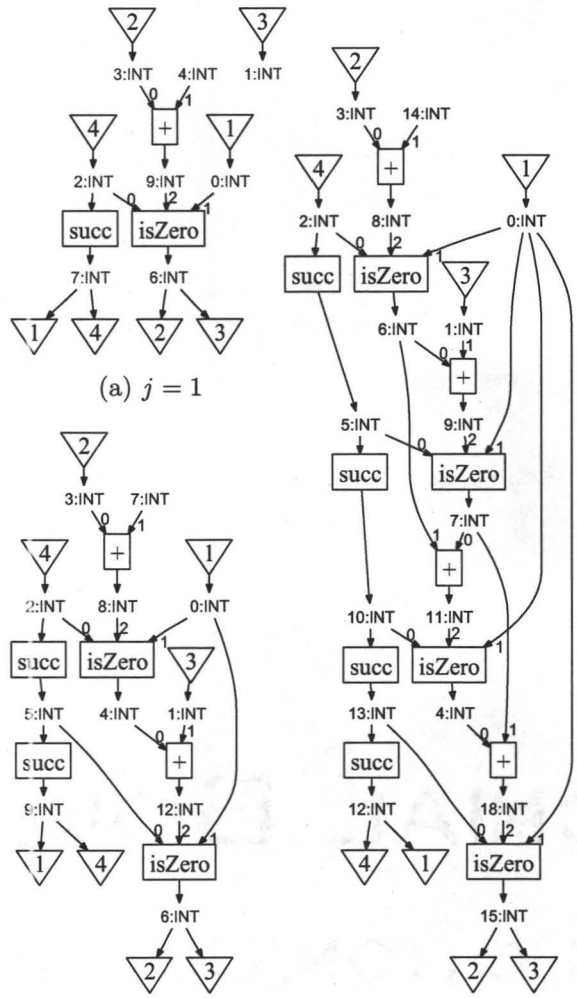
Figure 3.4$_{12}$ illustrates the definition for the base case which is decomposed into serial components. Figure 3.4(b)$_{12}$ shows disconnection of all input with loop distance greater than 1, together these are typed as B. As in first iteration, an input depending something from two or more iterations ago is undefined; we explicitly take this responsibility from user. In Fibonacci example Figure 3.1$_8$, if "counter" initially is assigned to 2 then in the first iteration "isZero" attempts to use constant input 1, which is undefined.



(a) $\mathbb{I}_K \otimes S_{\mathbf{G}}$ (b) $(\mathbb{I}_{K\otimes A} \otimes \mathsf{i}_B \otimes \mathbb{I}_{B'})$;$[\![\mathbf{G}']\!]_{j-1}$;$(\mathbb{I}_{K\otimes A} \otimes \mathbb{I}_B \otimes !_{B'})$ (c) $(\mathbb{I}_K \otimes S^{\sim}_{\mathbf{G}})$;$G$

Figure 3.5: $[\![\mathbf{G}]\!]_j$, $j > 1$

Figure 3.5$_{12}$ depicts the inductive part in the same manner as Figure 3.4$_{12}$. They only differ in (b) from each other. From a loop specification view, according to the definition of "loop specification" $\mathbf{G}'$, the output nodes typed by B actually get the value from input node typed by B', which has loop distance $d-1$ assuming there is only one element in B and original loop



(a) $j = 1$

(b) $j = 2$

(c) $j = 4$

Figure 3.6: Examples of $[\![\text{Fibonacci}]\!]_j$

distance is d, also passing from B' to B in $\mathbf{G}'$ takes 1 iteration, then input node typed by B gets the value generated d iterations ago from output node typed by B'. Again according to the definition of $\mathbf{G}'$, output node typed by B' is output node typed by B from shuffled $\mathbf{G}$. The semantics of $\mathbf{G}'$, $[\![\mathbf{G}']\!]$, preserves the meaning. Therefore the input node with type B in shuffled G gets what it needs.

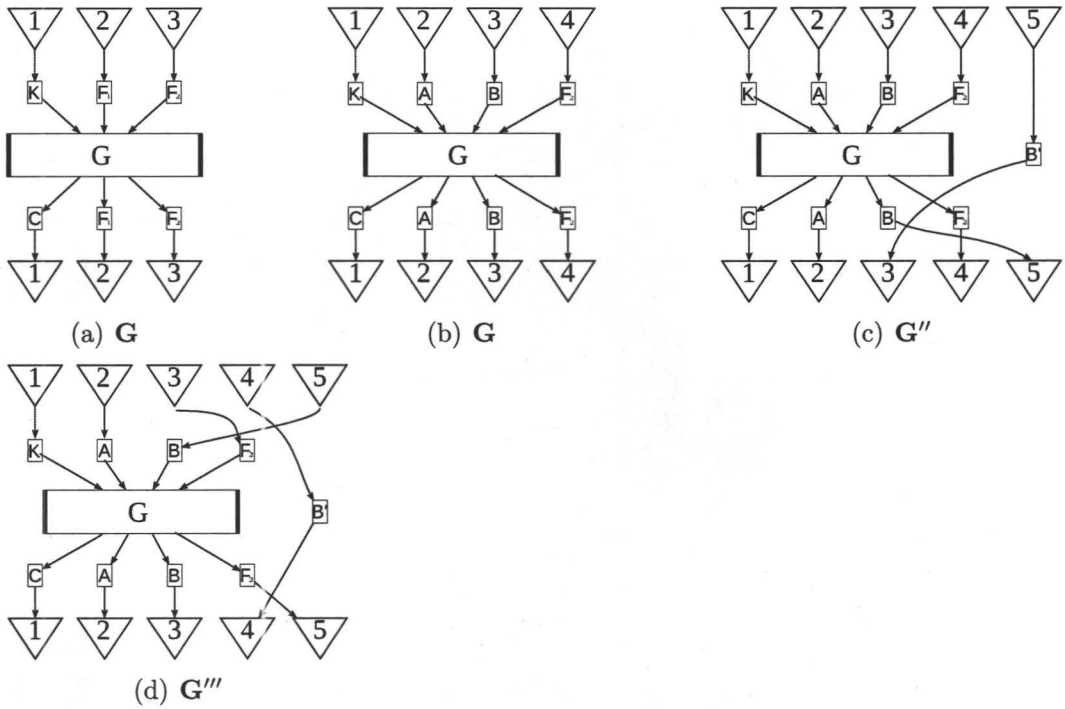


Figure 3.7: Steps to $\mathbf{G'''}$

Three examples in Figure 3.6_{13} are semantics for one, two, four iterations of loop specification of Fibonacci. Node identifiers on the left of colons are exposed to make referring easier. Figure $3.6(a)_{13}$ illustrates the semantics definition of the base case. The isolated input 3 is due to the loop distance of input 3 is greater than 1. In Figure $3.6(b)_{13}$ input 3 is consumed by an edge from second iteration. In Figure $3.6(c)_{13}$ node 6 and 7 are generated in iteration 1 and 3 and consumed in iteration 2 and 4, respectively. Input 1 is the only constant in the loop specification; in all three code graphs, the constant is fed to all edges from different iterations. In these three examples all edges are connected to correct nodes required by the original loop specification; therefore they satisfy the loop specification of Fibonacci. They are generated by a Haskell implementation of Def. 3.2_{10}.

3.3 Simplification

In this section we present another translation, from loop specification to loop specification; this translation reduces all loop distances which are greater than 1 to 1.

In Def. 3.2_{10}, it is shown that loop distances are reduced as recursive calling continues and eventually all loop distances become to 1. $\mathbf{G}'$ is defined based on $\mathbf{G}$, where all loop distance of inputs are decreased by 1 if the originals are greater than 1. A similar technique can be used to reduce all loop distances to 1 in general.

Let Simp() be the simplification function. It accepts a loop specification $\mathbf{G}$ with a decomposition on F, such that $G : K \otimes F \rightarrow C \otimes F$ where $F = F_1 \otimes F_2$. F_2 is an auxiliary interface, these inputs and outputs help to reduce loop distance; for example, B' in Def. 3.2_{10} is an auxiliary interface; initially, F_2 is equal to $\mathbb{1}$, the unit object of $\otimes$. S_G from Def. 3.1_9 is used to separate F_1 into two parts, $A \otimes B$, only, and all, inputs with loop distance greater than 1 are in B. As similar to $\mathbf{G}'$ in Def. 3.2_{10}, parallel composing G with $\mathbb{I}_{B'}$ is followed by swapping input 3 and 5; then the loop distances of inputs in B is 1 and inputs in B' are one less than their counterparts in the original B. Let $\mathbf{G}''$ be the loop specification represented by Figure $3.7(\mathrm{c})_{14}$.

The loop distance of inputs in A and B in $\mathbf{G}''$ are all equal to 1, but in general there are still some inputs with loop distances greater than 1 only in B'. Therefore a tail recursive call of Simp() is needed. Squeezing input/output 5 between 2 and 3 as in Figure $3.7(\mathrm{d})_{14}$ matches the interface required by Simp(); the new interface is $K \otimes (A \otimes B') \otimes (B \otimes F_2)$, $A \otimes B'$ and $B \otimes F_2$ match F_1 and F_2 respectively in the next call of Simp(). Another reason to transfer Figure $3.7(\mathrm{c})_{14}$ to Figure $3.7(\mathrm{d})_{14}$ is that the interface of a simplified loop specification suggests which part of interface should receive initial inputs. Simp() terminates when S_G does not decompose F_1, such that $S_G : A \otimes \mathbb{1}$.

Definition 3.3 *Formally* Simp() *can be defined as following, Let* $\mathbf{G}$ *be an input of* Simp(), *such that* $G : K \otimes F_1 \otimes F_2 \rightarrow C \otimes F_1 \otimes F_2$; *and* $S_{F_1} : F_1 \rightarrow A \otimes B$.

$$Simp(\mathbf{G}) = \begin{cases} \mathbf{G} & \text{if } B = \mathbb{1} \\ \mathbf{G}''' & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} G''' &: K \otimes A \otimes B' \otimes B \otimes F_2 \rightarrow C \otimes A \otimes B \otimes B' \otimes F_2 \\ G''' &= (\mathbb{I}_{K \otimes A} \otimes \mathbb{X}_{B \otimes F_2, B'}) ; (G \otimes \mathbb{I}_{B'}) ; (\mathbb{I}_{C \otimes A \otimes B} \otimes \mathbb{X}_{F_2, B'}) \\ \boldsymbol{d}'''(B') &= \boldsymbol{d}(B) - 1 \\ \boldsymbol{d}'''(A \otimes B \otimes F_2) &= 1 \end{aligned}$$

□

A simplified loop specification only contains the inputs with loop distance equal to 1, then **d** can be omitted. Def. 3.1_9 clearly shows that S_G separates all and only inputs with loop distance greater than 1 to B. Each call of Simp() splits F_1 to A and B, then redefine loop distance of B to 1 and combine it to F_2. Eventually when Simp() terminates, it means there is no input with loop distance greater than 1.

Loop specification abstracts over implementations of how to achieve loop distance. A simplified loop specification implements loop distance through loop specification transformation. Practically, the "bypass" inputs introduced in simplified loop specification cost more registers; this method is used in [CBS96] and COCONUT project. Here we try to formally present it, but we only did half of it. It is semi-formal since here is only a formal definition of simplification but without showing semantics preserving before and after transformation. The difficulty is that the original interface changes after simplification.

3.4 Wrap into RWCFG

The simplification function (Section 3.3_{15}) is one level less abstract than the general loop specifications, since it specifies how a loop distance which is greater than 1 can be translated to loop distances which are equal to 1. In this section, we show a way to implement a simplified loop specification, which is one more level less abstract; hope this can help reader to gain some intuitive understanding of loop specification.

In Figure 3.8, a loop specification is wrapped into Relation Weighted Control Flow Graph (RWCFG); RWCFG is control flow graph with relations as edges, it is formally introduced in Chapter 5_{37}. The middle graph is a control flow graph and the left one exposes the detail of each edge. Each edge of the left graph is a code graph. As introduced in Chapter 2_3, strict gs-monoidal category allows us to algebraically construct code graphs.

- $E = !_K \otimes \mathbb{I}_F \otimes \mathsf{EXIT}$, where $\mathsf{EXIT} : C \to C$ and $\mathsf{EXIT} \subseteq \mathbb{I}_C$.
- $T = \mathbb{I}_K \otimes \mathbb{I}_F \otimes (\mathsf{TIXE} ; !_C)$, where $\mathsf{TIXE} : C \to C$ and $\mathsf{TIXE} = \mathbb{I}_C \cap \overline{\mathsf{EXIT}}$.
- $L = (\nabla_K \otimes \mathbb{I}_F) ; (\mathbb{I}_K \otimes \mathsf{BODY})$, where BODY is a loop specification.

E checks the terminating condition of a loop. EXIT is a subidentity of C, it also bypasses the loop carried values of type F whereas it blocks all constants since they are independent of the loop. T is taken whenever the terminating condition is not satisfied. TIXE is a subidentity as well, it defines all entries not

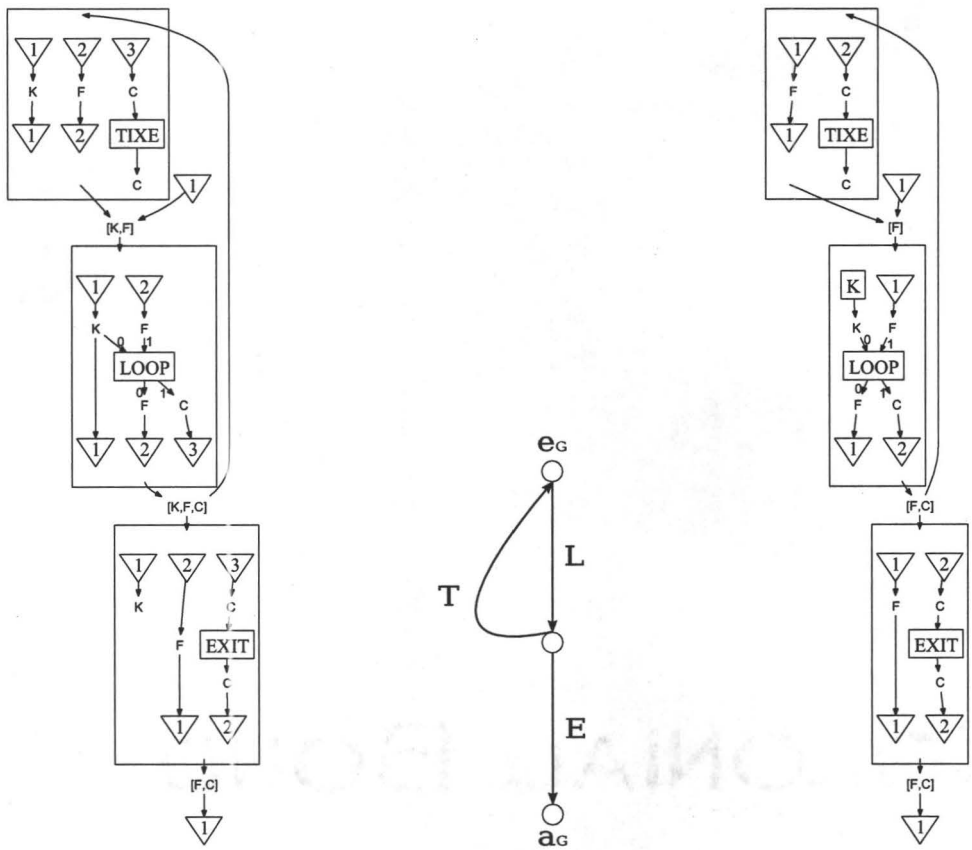


Figure 3.8: Wrap Loop Specification into RWCFG

defined in EXIT. It only transmits K and F to the next iteration. L embeds a loop specification G, and bypasses all constants. The Kleene representation is $L_{\S}(T_{\S}L)^{*}{}_{\S}E$. If the constants are not relevant, then they can be hidden in L by adopting co-termination i_K. As illustrated on right in Figure 3.8, the result RWCFG can be more concise.

- $E = \mathbb{I}_F \otimes \mathsf{EXIT}$, where $\mathsf{EXIT} : C \to C$ and $\mathsf{EXIT} \subseteq \mathbb{I}_C$.
- $T = \mathbb{I}_F \otimes (\mathsf{TIXE}_{\S}!_C)$, where $\mathsf{TIXE} : C \to C$ and $\mathsf{TIXE} = \mathbb{I}_C \cap \overline{\mathsf{EXIT}}$.
- $L = ((\mathsf{i}_K{}_{\S}K) \otimes \mathbb{I}_F)_{\S}\mathsf{BODY}$, where BODY is a loop specification.

Chapter 4

Construction of Loop Spec.

In Chapter 3_7, we mainly discussed operational semantics, simplification and embedding loop specifications into RWCFGs. In this chapter, a new structure *Extensible Loop Specification* is introduced, which is formally defined based on loop specification. A loop specification represents an entire loop body, but an extensible loop specification is mainly used to represent a piece of a loop body; an extensible loop specification can also represent an entire loop body, this is a special case. Chapter 3_7 is important that we would like to build the new structure, *extensible loop specification*, on the top of something rigorous; loop specification has a formal syntax but informal semantics, in Chapter 3_7 we present a way to translate loop specifications to something having formal syntax and formal semantics; in our case loop specifications are translated to infinite sets of code graphs.

A definition of extensible loop specification is presented in Section 4.1_{19}, then composition is defined in Section 4.2_{20}; in Section 4.3_{23}, an application shows some benefits of using extensible loop specification; motivation and steps of parameterised decomposition are in Section 4.4_{26}

4.1 Extensible Loop Spec.

An extensible loop specification includes a new part in the input and output interfaces of loop specification, as following;

$$K \otimes F \otimes E_{in} \to C \otimes F \otimes E_{out}$$

Unlike F, the interfaces E_{in} and E_{out} are not equipped with loop distances; they are intended to interface with other compatible extensible loop specification. This new structure mixes loop specifications and code graphs; it inherits

the properties of both; through E_{in} and E_{out}, composing extensible loop specification is similar to serial composing code graphs, loop specification behavior is by the fact that F are assigned with loop distances.

Definition 4.1 *An extensible loop specification is a tuple* $(G, \boldsymbol{d}, E_{in}, E_{out})$, G *is a code graph with signature:*

$$G : K \otimes F \otimes E_{in} \rightarrow C \otimes F \otimes E_{out}$$

if G' *is defined by swapping* F *and* E_{in}, F *and* E_{out}; *then* $(G', \boldsymbol{d})$ *is a loop specification, where* $K' = K \otimes E_{\text{in}}$ *and* $C' = C \otimes E_{\text{out}}$. □

In Def. 4.1_{20}, an extensible loop specification syntactically becomes a loop specification if E_{in} and E_{out} are combined into K and C respectively. Then it can be implied that the loop carried values must be generated and consumed in the same extensible loop specification. The advantages and disadvantages will be discussed in later sections. In two extreme cases, F is empty or E_{in} and E_{out} are empty, extensible loop specification becomes a code graph or a loop specification respectively. Two different extreme cases are when only E_{in} or only E_{out} is empty; these are called right-extensible or left-extensible loop specification.

left-extensible $K \otimes F \otimes E_{in} \rightarrow C \otimes F$

right-extensible $K \otimes F \rightarrow C \otimes F \otimes E_{out}$

Extensible loop specification is a closed and more general domain for composing and decomposing operations of loop specification. Also it introduces the "module" concept to loop specifications, users can arrange semantically related edges into one extensible loop specification; it not only improves organization but may increases reusability.

4.2 Composition

A new extensible loop specification can be built by composing two extensible loop specifications; when composing them, the extended interface E_{in} and E_{out} are serially composed whereas the other parts of interface are composed in parallel, the last step is to sort the new interface elements into standard extensible loop specification interface. Similar to function composition, composition of extensible loop specification is not communitive but associative.

Definition 4.2 *Let* $A = (G_A, \boldsymbol{d}_A, E_{A\,in}, E_{A\,out})$, $B = (G_B, \boldsymbol{d}_B, E_{B\,in}, E_{B\,out})$ *be two extensible loop specifications,*

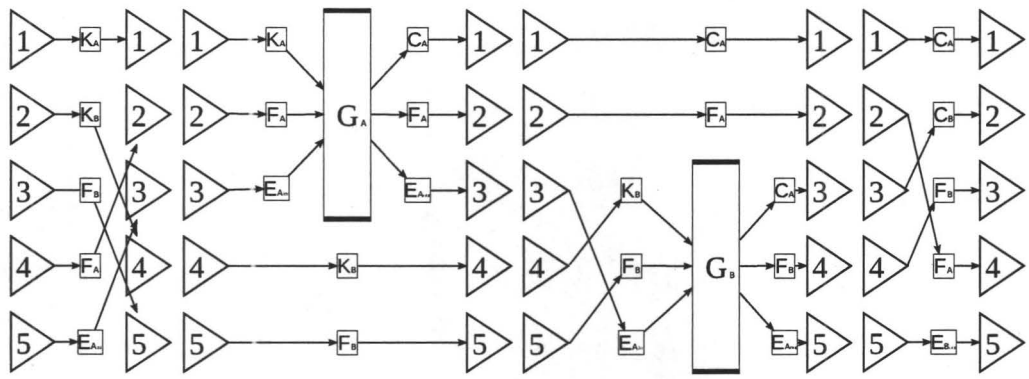
- $G_A : K_A \otimes F_A \otimes E_{A\,in} \rightarrow C_A \otimes F_A \otimes E_{A\,out}$
- $G_B : K_B \otimes F_B \otimes E_{B\,in} \rightarrow C_B \otimes F_B \otimes E_{B\,out}$

$A \circledast B$ *is defined if* $E_{A\,out} = E_{B\,in}$.
Let $C = A \circledast B$, *then* C *is defined as,*

- $G_C : (K_A \otimes K_B) \otimes (F_B \otimes F_A) \otimes E_{A\,in} \rightarrow (C_A \otimes C_B) \otimes (F_B \otimes F_A) \otimes E_{B\,out}$
  $G_C = A_1 ; A_2 ; A_3 ; A_4$
  - $A_1 = \mathbb{I}_{K_A} \otimes \mathbb{X}_{F_A \otimes E_{A\,in}, K_B \otimes F_B}$
  - $A_2 = G_A \otimes \mathbb{I}_{K_B \otimes F_B}$
  - $A_3 = \mathbb{I}_{C_A \otimes F_A} \otimes (\mathbb{X}_{K_B \otimes F_B, E_{B\,in}} ; G_B)$
  - $A_4 = \mathbb{I}_{C_A} \otimes \mathbb{X}_{F_A, C_B \otimes F_B} \otimes E_{B\,out}$
- $E_{C\,in} = E_{A\,in}$
- $E_{C\,out} = E_{B\,out}$
- $\boldsymbol{d}_c = \boldsymbol{d}_B +\!\!+ \boldsymbol{d}_A$ □

The above formally specifies how two extensible loop specifications are composed. A_2 right-extends G_A with loop specification aspect of interface of G_B, G_B is extended to the left with an appreciated interface after relocating $E_{B\,in}$ to chase $E_{A\,out}$ by swapping $K_B \otimes F_B$ and $E_{B\,in}$. A_1 and A_2 shuffle input, respectively, and output interface of $A_2 ; A_3$ to match desired signature. When G_A is left-extensible then G_B must be right-extensible in order to make $A \circledast B$ defined; in this case composition of extensible loop specification becomes parallel composition of two loop specifications with shuffled interface. In another case if all loop specification parts are all empty, such that $F_A = K_A = C_A = \mathbb{1} = C_B = K_B = F_B$, then the composition becomes serial composition of code graph. Figure 4.1$_{22}$ visually presents the implementation of composition based on Def. 4.2$_{21}$

Associativity of composition can be checked by expanding the left and right hand sides according Def. 4.2$_{21}$. When proving equality of two sides, functoriality will be used to distribute identity over serial compositions.



(a) A_1 (b) A_2 (c) A_3 (d) A_4

Figure 4.1: Composition of Extensible Loop Specification

Lemma 4.1 *There exist a unique left and a unique right identity of* $\circledast$ *to every extensible loop specification. Let* $X = (G_X, \boldsymbol{d}_X, E_{X\,in}, E_{X\,out})$ *and* $G_X : K_X \otimes F_X \otimes E_{X\,in} \rightarrow C_X \otimes F_X \otimes E_{X\,out}$. *The left and the right identity are* $U_L = (\mathbb{I}_{E_{in}}, \varnothing, E_{U_L in}, E_{U_L in})$ *and* $U_R = (\mathbb{I}_{E_{out}}, \varnothing, E_{U_R in}, E_{U_R out})$, *respectively; such that,*

$$U_L \circledast X = X = X \circledast U_R$$

Proof:
There exists a left identity, $U'_L = (G_{U'_L}, \varnothing, E_{U'_L in}, E_{U'_L out})$.Let the signature of U'_L be $K_{U'_L} \otimes F_{U'_L} \otimes E_{U'_L in} \rightarrow C_{U'_L} \otimes F_{U'_L} \otimes E_{U'_L out}$. Since $G_{U'_L} \circledast X = X$, then $K' = F' = C' = \mathbb{1}$ and $E_{U'_L in} = E_{U'_L out} = E_{X\,in}$ according to Def. 4.2_{21}. Then the signature of U'_L is simplified to $E_{in} \rightarrow E_{in}$. Then A_2 in Def. 4.2_{21} is equal to $G_{U'_L} \otimes \mathbb{I}_{K \otimes F}$, which is expected to be an identity. Since the bifunctor $\otimes$ preserves identities, $G_{U'_L}$ must be equal to $\mathbb{I}_{E_{in}}$. □

Unfortunately, composition and identities are defined but extensible loop specification still does not form a category. The reason is that let f and g be tow extensible loop specification, also $f : \mathcal{A} \rightarrow \mathcal{B}$ and $g : \mathcal{B} \rightarrow \mathcal{C}$, in general $f \circledast g$ is *not* typed to $\mathcal{A} \rightarrow \mathcal{C}$; this is a condition to be a category.

In the next example(Figure 4.2_{23}), we show how extensible loop specification can be used to organize loop body into pieces, where each piece should contain semantically related edges. Fibonacci loop specification is made up by two pieces, COUNTER and ADDER. COUNTER is right-extensible and ADDER

is left-extensible, a loop specification is naturally obtained by composing two. Pink hollow arrows indicate E_{in} and E_{out}. Figure 4.2(a)$_{23}$ is the "loop overhead"; it maintains a loop counter; it does not need any constant input, it returns counter as control information through output 1. Now it can be considered as a separated "module" providing loop counter service. It only allows other parts to read the counter through its extended interface, output 3 in this case. Moreover, it is a better loop counter than `for (i=a; i < b; i++)` in language C. In C the counter `i` can be overwritten in loop body whereas the counter in Figure 4.2(a)$_{23}$ will only be updated in its own "module". This shows one advantage of having loop specification behavior of extensible loop specification.

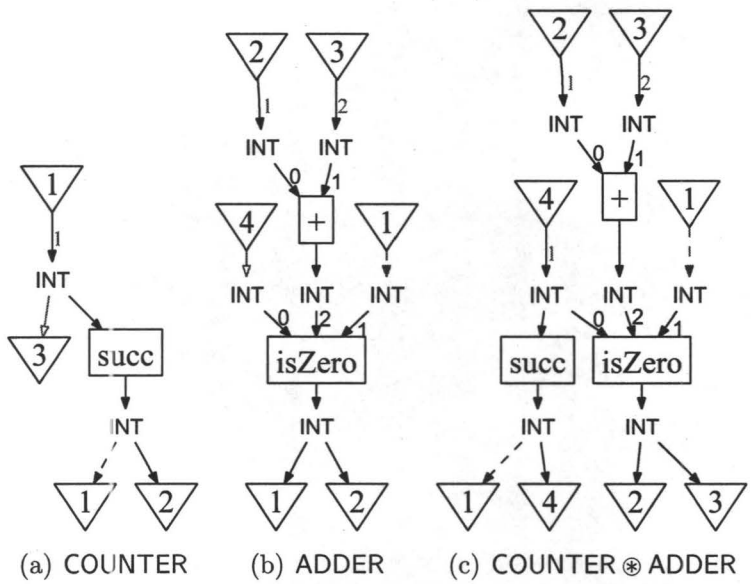


Figure 4.2: Decomposed Fibonacci

4.3 A Motivating Example

Before introducing parameterised decomposition, we present an example to show some advantages of extensible loop specification. The example is a kernel of a scientific computation developed in COCONUT. The kernel is presented in nested graphs, the outer graphs are control flow and the inner graphs are code graphs(Chapter 2$_3$). In [AK08a], optimized implementation of the kernel can be achieved by a sequence of graph transformation. All edges in the inner graphs are pseudo instructions.

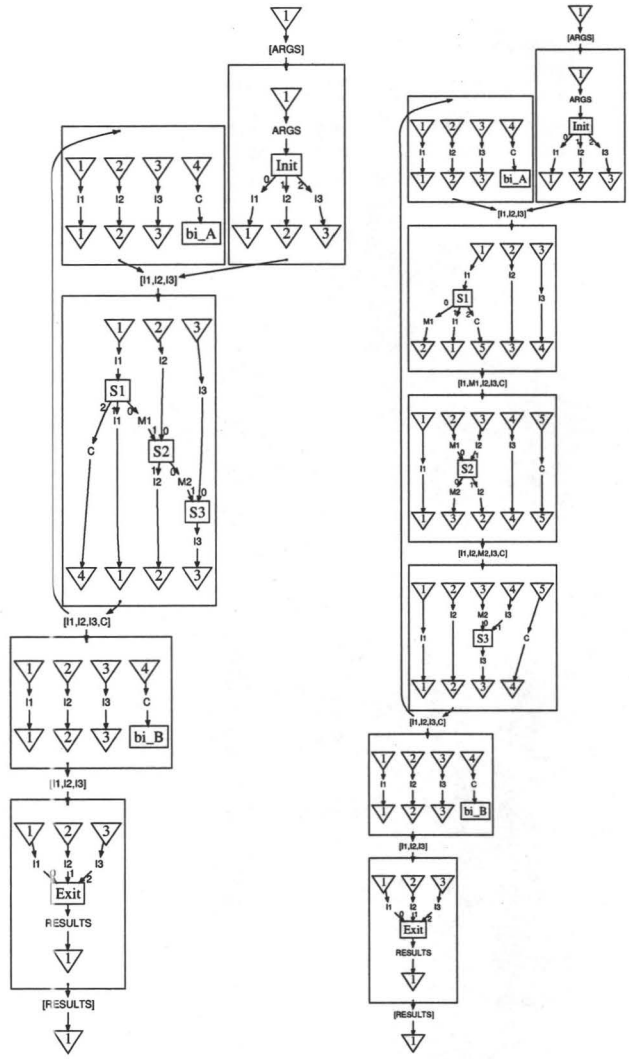


Figure 4.3: Motivating Example

The outer graph in Figure $4.3(a)_{24}$ is a loop, the loop body is the (outer) edge containing edge "S1","S2" and "S3". In Figure $4.3(b)_{24}$, the loop body is splitten into three (outer) edges, this is a just code graph decomposition. Next we show a better representation based on extensible loop specifications.

Figure 4.3_{25} is the loop body from Figure $4.3(b)_{24}$. Figure 4.5_{26} is the

same loop body but represented by extensible loop specification. In the top edge in Figure 4.3_{25} output 1 actually is a loop carried dependency, since the middle and bottom edges just bypass it. Figure 4.5_{26} shows a more clear representation, such that in the most left figure in Figure 4.5_{26} output 2 is a loop carried dependency, which is consumed by edge $S1$ in the next iteration. The beneficial of using extensible loop specification is that it can express loop carried dependency, then the output 2 does not have to pass into other outer edges.

Another advantage is a result of the previous one, only information needed in the next stage is passed to the next stage; since less information is passed, then the representation becomes more concise.

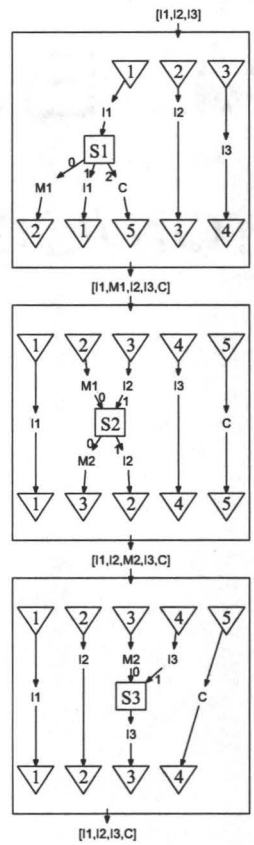


Figure 4.4: Code graph representation

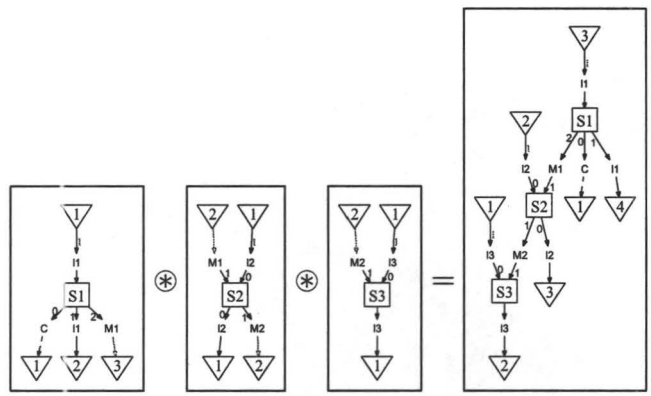


Figure 4.5: Extensible loop specification representation

4.4 Parameterised Decomposition Algorithm

Figure 4.2_{23} is used as an example of composition, but Figure $4.2(a)_{23}$ and Figure $4.2(b)_{23}$ are generated by an parameterised decomposition algorithm applied on Figure $4.2(c)_{23}$. In this section we present the algorithm for decomposing an extensible loop specification.

4.4.1 Motivation

Developing decomposition operation is partially inspired by Decoupled Software Pinpelinning(DSWP). DSWP decomposes a loop into several parts, it is intended to execute a loop by multi-threads [ORSA05] and to improve fetch of recursive data structure[RVVA04]. DSWP uses an alternative graph representation for loops, instructions are denoted by nodes and edges represent instruction dependency and loop-carried dependency. Briefly, the DSWP algorithm[ORSA05] discovers strongly connected components as basic units, these basic units form a direct acyclic graph; then distributing basic units into different threads. In this section we present a fundamental tool for decomposing an extensible loop specification; such decomposition can potentially be used for extracting multi-threads from existing loops. Ideally, if a loop X is decomposed into $X_1 \circledast \ldots \circledast X_n$, then each X_i can be distributed to a thread. Inter-thread communication is needed for each composed extensible interface; moreover, inserting buffers for each inter-thread communication allows differ-

ent parts to execute in different logic iteration. Extensible loop specification is an algebraic treatment to loop manipulation with formal semantics. Multi-thread is a complex scheduling problem, it is not in the scope of this paper. The basic decomposition algorithm we developed is deterministic and is intended to be used by scheduling algorithms.

4.4.2 Overview

Decomposition is a closed unary operator on extensible loop specification. An ideal decomposition would be completely reverse of composition, such that X can be decomposed to X_L and X_R and $X = X_L \circledast X_R$. Without any hint, decomposition is not unique, a quick proof is that: let X_L or X_R be left or respectively right unit of $\circledast$, these are two decompositions. In order to determine a unique decomposition, user can specify constraints through two different steps. The first step is to allow user specify a unique interface decomposition as parameter. Let $X = (G_X, \mathbf{d}_X, E_{X\,in}, E_{X\,out})$,

$$G_X : K \otimes F \otimes E_{X\,in} \rightarrow C \otimes F \otimes E_{X\,out}$$

then assuming the following interface decomposition is specified by user, but E'_{in} and E'_{out} are not specified by user since they are not exposed in the interface of G_X.

$$G_{X_L} : K_L \otimes F_L \otimes E_{X\,in} \rightarrow C_L \otimes F_L \otimes E'_{out}, \tag{4.1}$$

$$G_{X_R} : K_R \otimes F_R \otimes E'_{in} \rightarrow C_R \otimes F_R \otimes E_{X\,out}. \tag{4.2}$$

Let X_L and X_R be two extensible loop specifications, they are *intended* to decompose X. When composing two extensible loop specifications, the right operand consumes the value generated in left operand by gluing E_{in} and E_{out}. This means the direction of dependency is from left to right; therefore edges in the left operand can not consume any value generated in right operand. Further, if an edge depends on a node in right operand then this edge must be in the right operand; another conclusion about left operand is that if an edge is depended in left operand then it must be in the left operand. Following this idea, for a given X and a given interface decomposition, there exist the "smallest" X_L and the "smallest" X_R, and they contain all and only edges which must be belong to them. In the sense of "smallest", it means that in every other decomposition of X, namely X'_L and X'_R, X'_L can be obtained by adding more edges and nodes to X_L, so can X'_R to X_R. In general $X_L \circledast X_R \neq X$, there exists another extensible loop specification connecting X_L and X_R, namely X_M.

$$X_M : E'_{out} \rightarrow E'_{in}$$

Strictly speaking, X_M is a code graph since loop specification aspect disappears as $K_M = F_M = C_M = \mathbb{1}$. In the first step, with a user given interface decomposition, X is decomposed into three parts, X_L, X_M, X_R,

$$X = X_L \circledast X_M \circledast X_R. \tag{4.3}$$

Equ. 4.3_{28} implicitly requires completeness and disjointness of edge sets of three parts,

$$\mathcal{E}_{X_L} \cup \mathcal{E}_{X_M} \cup \mathcal{E}_{X_R} = \mathcal{E}_X \text{ and } \mathcal{E}_{X_L} \cap \mathcal{E}_{X_M} \cap \mathcal{E}_{X_R} = \varnothing$$

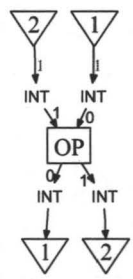


Figure 4.6: X

In some cases, extensible loop specifications can not be decomposed by given interface decompositions; Figure $4.4.2_{28}$ is a simple counter example, when decomposing X with a given interface decomposition $F_L = 1$ and $F_R = 2$, the edge "OP" must be in X_R since it consumes input 2, also it must be in X_L since output 1 depends on it; therefore this decomposition is impossible. A checking method to determine decomposable will be mentioned when presenting algorithm of computing X_L and X_R.

E_{in} from X is not allowed to be partially included into X_R, neither is E_{out} to X_L. E_{in} becomes extensible input of X_L and E_{out} becomes extensible output of X_R. This decision is influenced by how composition is defined; if both E_{in} and E_{out} were decomposed to left and right parts, then they would have had to make new interface components; the reasons they can not join existing interface components are,

F	there is no loop distance assigned to E_{in} and E_{out}.
K or C	makes the signature of $X_L \circledast X_M \circledast X_R$ different from X, since part of E_{in} and E_{out} is combined into K and C respectively.
E'_{in} or E'_{out}	makes $X_L \circledast X_M \circledast X_R$ undefined in general, since new E'_{in} and new E'_{out} are not equal.

4.4.3 First Step of The Algorithm

The algorithm is presented in imperative manner; it is given a extensible loop specification X and an interface decomposition Equ. 4.1_{27} and Equ. 4.2_{27}; $\mathcal{N}$ and $\mathcal{E}$ are the node sets and edge set in G_X. The algorithm returns X_L, X_M, X_R if given inputs are decomposable otherwise it does not return anything. When $\mathcal{N}$ or $\mathcal{E}$ is subscripted, then it is a subset of $\mathcal{N}$ or $\mathcal{E}$ induced by the subscription, a subscript can be a list of elements in $\mathcal{N}$ or $\mathcal{E}$, or it can be a list of reference to $\mathcal{N}$, like K, F, C, E. If $\mathcal{N}$ or $\mathcal{E}$ appears as a subscript, then it acts like a "type" indicating where all list elements are from. When $\mathcal{N}$ or $\mathcal{E}$ is promoted from subscript to host, then it is a conversion from list to set; the other direction does not work. In the decomposition algorithm, src' and trg' are variants of src and trg introduced in Def. 2.1_3;

$$\mathsf{src}' : \mathbb{P}^{\mathcal{E}} \to \mathbb{P}^{\mathcal{N}}, \qquad \mathsf{trg}' : \mathbb{P}^{\mathcal{E}} \to \mathbb{P}^{\mathcal{N}},$$
$$\mathsf{src}'(es) = \{n \mid e \in es, n \in \mathsf{src}(e)\}, \qquad \mathsf{trg}'(es) = \{n \mid e \in es, n \in \mathsf{trg}(e)\}.$$

1. - $\mathcal{N}_{G_{X_R}} \leftarrow \mathcal{N}_{F_{in\,R}} \cup \mathcal{N}_{K_R} \cup \mathcal{N}_{C_R} \cup \mathcal{N}_{F_{out\,R}} \cup \mathcal{N}_{E_{out}}$
   - $\mathcal{N}_{G_{X_L}} \leftarrow \mathcal{N}_{F_{out\,L}} \cup \mathcal{N}_{C_L} \cup \mathcal{N}_{K_L} \cup \mathcal{N}_{F_{in\,L}} \cup \mathcal{N}_{E_{in}}$
2. - $(\mathsf{visitedR}_{\mathcal{N}}, \mathsf{visitedR}_{\mathcal{E}}) \leftarrow \mathit{breadth\_first\_search}(G_X, \mathcal{N}_{G_{X_R}})$
   - $(\mathsf{visitedL}_{\mathcal{N}}, \mathsf{visitedL}_{\mathcal{E}}) \leftarrow \mathit{reverse\_breadth\_first\_search}(G_X, \mathcal{N}_{G_{X_L}})$
3. if $\mathcal{E}_{\mathsf{visitedR}} \cap \mathcal{E}_{\mathsf{visitedL}} \neq \varnothing$ then X_L and X_R are **not** found.
4. $\mathcal{E}'_{G_{X_R}} \leftarrow \mathcal{E}_{G_X} - \mathcal{E}_{\mathsf{visitedR}}$
5. $\mathcal{N}_{\mathsf{argR}} \leftarrow \mathsf{src}'_{G_X}(\mathcal{E}_{\mathsf{visitedR}})$
6. $\mathcal{N}_{G_{X_R}} \leftarrow \mathcal{N}_{G_{X_R}} \cup \mathcal{N}_{\mathsf{argR}}$
7. $E'_{in\mathcal{N}} \leftarrow \mathit{srot}(\mathcal{N}_{G_{X_R}} - \mathcal{N}_{\mathsf{visitedR}})$
8. $\mathcal{N}'_{G_{X_R}} \leftarrow \mathcal{N}_{G_X} - \mathcal{N}_{G_{X_R}}$
9. $G'_{X_R} \leftarrow \mathit{remove\_edges}(G_X, \mathcal{E}'_{G_{X_R}})$
10. $G_{X_R} \leftarrow \mathit{remove\_nodes}(G'_{X_R}, \mathcal{N}'_{G_{X_R}})$
11. $X_R \leftarrow (G_{X_R}, \mathbf{d}_X)$
12. $\mathcal{E}'_{G_{X_L}} \leftarrow \mathcal{E}_{G_X} - \mathcal{E}_{\mathsf{visitedL}}$
13. $\mathcal{N}_{\mathsf{argL}} \leftarrow \mathsf{trg}'_{G_X}(\mathcal{E}_{\mathsf{visitedL}})$
14. $\mathcal{N}_{G_{X_L}} \leftarrow \mathcal{N}_{G_{X_L}} \cup \mathcal{N}_{\mathsf{argL}}$
15. $\mathcal{N}_{E'_{in}1} \leftarrow \mathcal{N}_{G_{X_L}} - \mathcal{N}_{\mathsf{visitedL}}$

16. $\mathcal{N}_{E'_{in}2} \leftarrow \mathcal{N}_{G_{X_L}} \cap (\mathsf{src}'_X(\mathcal{E}'_{G_{X_L}}) \cup (\mathcal{N}_{C_R} \cup \mathcal{N}_{F_{out\,R}} \cup \mathcal{N}_{E_{out}}))$
17. $E'_{in\mathcal{N}} \leftarrow sort(\mathcal{N}_{E'_{in}1} \cup \mathcal{N}_{E'_{in}2})$
18. $\mathcal{N}'_{G_{X_L}} \leftarrow \mathcal{N}_{G_X} - \mathcal{N}_{G_{X_L}}$
19. $G_{X_L} \leftarrow remove\_nodes(remove\_edges(G_X, \mathcal{E}'_{G_{X_L}}), \mathcal{N}'_{G_{X_L}})$
20. $X_L \leftarrow (G_{X_L}, \mathrm{d}_X)$
21. $\mathcal{N}'_{G_{X_M}} \leftarrow (\mathcal{N}_{G_{X_L}} \cup \mathcal{N}_{G_{X_R}}) - (\mathcal{N}_{E'_{in}} \cup \mathcal{N}_{E_{out}})$
22. $G_{X_M} \leftarrow remove\_nodes(remove\_edges(G_X, \mathcal{E}_{G_{X_L}} \cup \mathcal{E}_{G_{X_R}}), \mathcal{N}'_{G_{X_M}})$
23. $X_M \leftarrow (G_{X_M}, \varnothing)$

Explanation

In Line 1_{29}, $\mathcal{N}_{G_{X_R}}$ and $\mathcal{N}_{G_{X_L}}$ are node set of X_R and X_L as suggested in their subscript. Initially, they only contain the given interfaces of X_R and X_L, respectively. Line 2_{29} calculate all edges and nodes depending on X_R and reversely depending on X_L; in each case, it returns a list of nodes and a list of edges; nodes and edges are ordered by breadth first traversal. In this algorithm, X_L, x_R are obtained by deconstructing X, so the traversal algorithm only needs to discover reachable edges and nodes, then both breadth(BFS) and depth(DFS) first search are applicable here. However, if X_L, X_R are built in a constructive way, then all reachable nodes and edges also need to be in reverse topological order for X_R and topological order for X_R. then BFS should be used. Line 3_{29} checks whether X with the given interface decomposition is decomposable; if two edge sets are not disjointed then the algorithm reports fail otherwise keeps finding $E'_{in\mathcal{N}}$, $E'_{out\mathcal{N}}$, X_M.

In Line 5_{29}, $\mathcal{N}_{\mathsf{argR}}$ contains all the nodes consumed by edge set $\mathcal{E}_{\mathsf{visitedR}}$, if a node not in $\mathcal{N}_{\mathsf{visitedR}}$ is consumed by an edge in X_R then connection must be made through extended interface $E'_{out\mathcal{N}}$. Line 6_{29} updates $\mathcal{N}_{G_{X_R}}$ by including $\mathcal{N}_{\mathsf{argR}}$. In Line 7_{29}, $\mathcal{N}_{E'_{in}}$ is defined as all nodes are needed but not reachable from input interface of X_R; these also include a special case that elements in output interface are not generated in X_R. Compatibility between E'_{in} and E'_{out} can be easily obtained by sorting both $E'_{in\mathcal{N}}$ and $E'_{out\mathcal{N}}$ to $\mathcal{N}_{E'_{in}}$ and $\mathcal{N}_{E'_{out}}$. Using the *sort* function imposes a pre-condition on the algorithm that nodes must be linear ordering. E'_{out} is initially created as a set, one may ask that why there can not be duplicated elements in E'_{out}; by the definition of code graph it is possible to have two outputs referring to the same node; however one can imagine that the decomposition algorithm "splits" a node, so if two

edges consume a node then they "share" it in the body of G_{X_R} not in interface. Line 8_{29} - Line 11_{29} remove all nodes and edges which are not used in G_{X_R}.

Calculating X_L is *almost* dual to X_R, $\mathcal{N}_{G_{X_L}}$ is initiated with output of X_R rather than inputs, *reverse_breadth_first_search* returns lists of nodes and edges reversely depending on output. In Line 13_{29}, trg′ replaces its reverse counterpart src′. However, join-free property of code graph is not preserved when reversing the direction of all edges. In X_R, $\mathcal{N}_{E'_{in}}$ only contains all unsupported nodes, since a node can not be supported twice. In X_L, since a node can be used more than once, then $\mathcal{N}_{E'_{out}}$ not only contains all unused nodes but also used nodes in both G_{X_L} and outside, which is implemented by Line 16_{30}.

Line 21_{30} - Line 23_{30} calculate X_M, it is formed by two parts,

- the edges and nodes not included in either G_{X_L} or G_{X_R},
- $\mathcal{N}_{E'_{in}}$ and $\mathcal{N}_{E'_{out}}$.

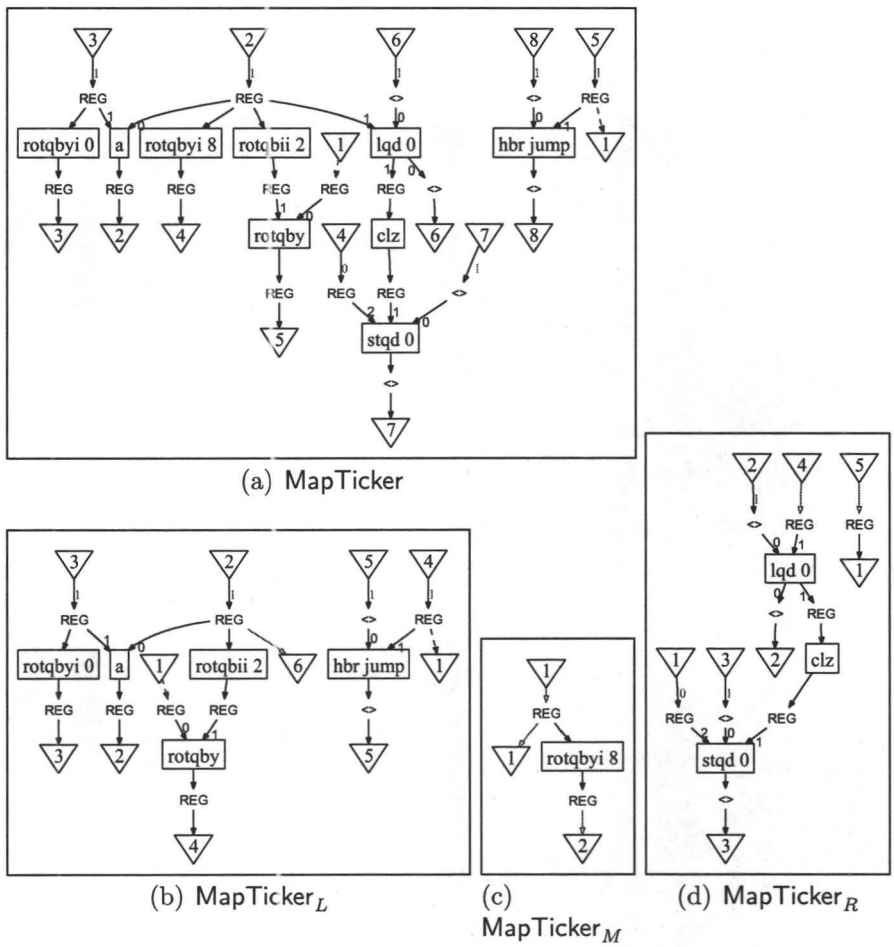
Example

MapTicker is the tricky loop overhead presented in [AK08b]; Figure 4.7_{32} is a loop specification implementation of it. In this implementation, edge "clz" is a dummy edge to pretend a pure one input and one output computation. Edges in Figure 4.7_{32} are SPU[1] instructions. Low level implementation is really not comfortable for documentation or reading, here we use the decomposition algorithm to organize semantically closed edges into "modules"; for example, edge "lqd 0", "clz", "stqd" can stay in the same module, then apply decomposition algorithm on Figure $4.7(\text{a})_{32}$ with decomposed interface $C_R = K_R = \varnothing$, $F_R = \{4, 6, 7\}$; Figure $4.7(\text{b})_{32}$, Figure $4.7(\text{c})_{32}$, Figure $4.7(\text{d})_{32}$ are results of the algorithm. MapTicker′ replaces the dummy edge "clz" by "sf", the later has two inputs and one output. In two decompositions we find that $\text{MapTikcer}_L = \text{MapTikcer}'_L$, this allows us to recognize reusability of MapTikcer_L.

4.4.4 Second Step of The Algorithm

The decomposition algorithm has not completed yet, it needs to further decompose X_M, for convenience let $M = X_M$, into left and right parts, M_L and

[1]The Synergistic Processor Unit (SPU) is part of the SPE in the Cell Broadband Engine Processor.



(a) MapTicker

(b) MapTicker$_L$

(c) MapTicker$_M$

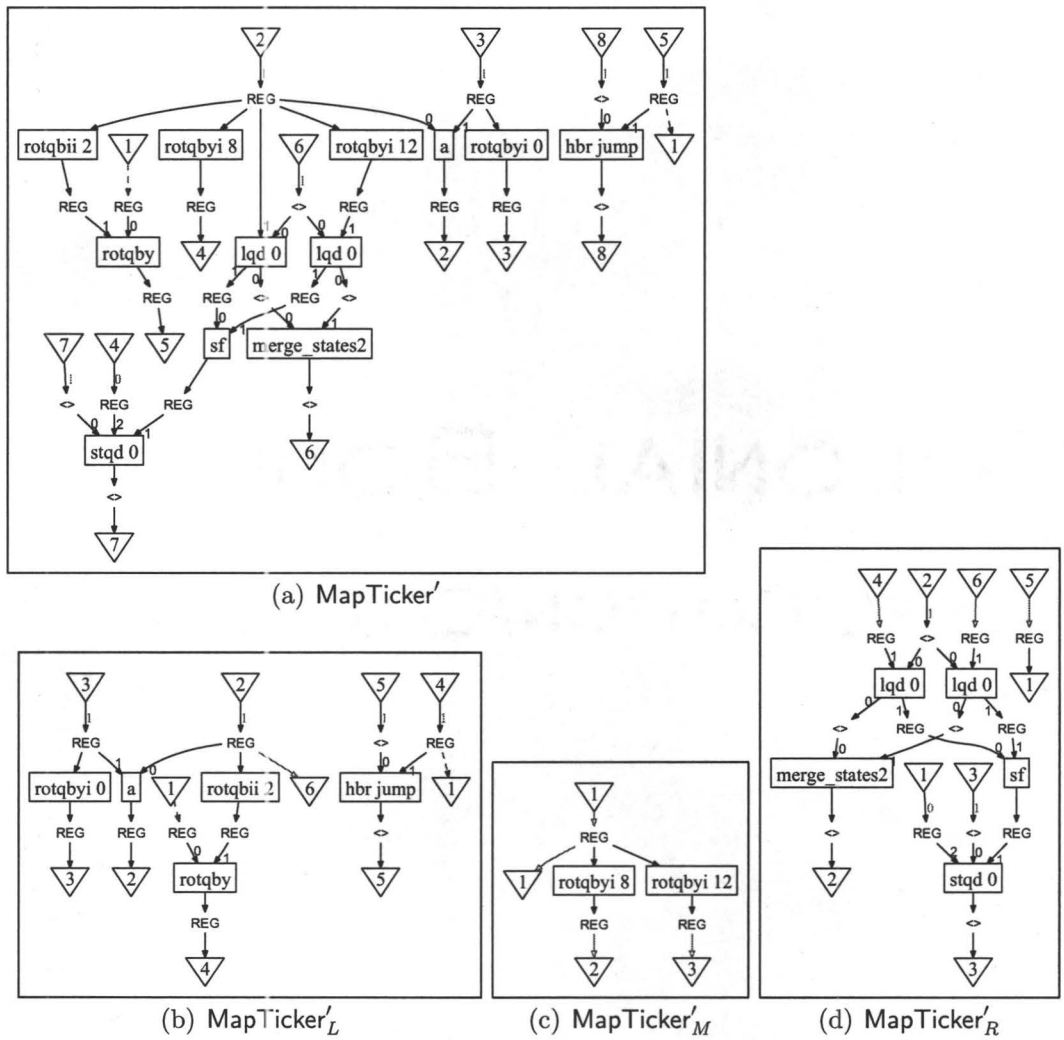
(d) MapTicker$_R$

Figure 4.7: Decomposition of MapTicker

M_R; then a completed decomposition is $X = (X_L \circledast M_L) \circledast (M_R \circledast X_R)$. Decomposing M requires the user to specify exactly how to split edge set of M into two parts.

$$M = M_L \circledast M_R, \text{where } M_L : E'_{out} \rightarrow H,\ M_R : H \rightarrow E'_{in}$$

The precondition is that the given extended loop specification contains only the extended interface, the other parts of interface are equal to $\mathbb{1}$; the given edge set $\mathcal{E}_{M_L}$ is intended to be used as edge set of M_L. In the following algorithm, $\mathcal{N}$ and $\mathcal{E}$ refers node set and edge set of G_M. Deconstructive method is used



(a) MapTicker′

(b) MapTicker′$_L$ (c) MapTicker′$_M$ (d) MapTicker′$_R$

Figure 4.8: Decomposition of MapTicker'

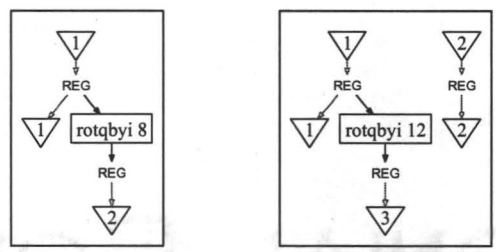
again to build M_L and M_R. Line 2_{33} checks possibility of decomposition of M with user specified edge set.

1. $\mathcal{E}_{G_{M_R}} \leftarrow \mathcal{E} - \mathcal{E}_{G_{M_L}}$
2. if $\mathsf{trg}'(\mathcal{E}_{G_{M_R}}) \cap \mathsf{src}'(\mathcal{E}_{G_{M_L}}) \neq \varnothing$ then M_L and M_R are *not* found.
3. $\mathcal{N}_{G_{M_L}} \leftarrow E'_{out.\mathcal{N}} \cup \mathsf{src}'(\mathcal{E}_{G_{M_L}}) \cup \mathsf{trg}'(\mathcal{E}_{G_{M_L}})$
4. $H_{\mathcal{N}} \leftarrow sort(\mathcal{N}_{G_{M_L}} \cap (\mathsf{src}'(\mathcal{E}_{G_{M_R}}) \cup E'_{in\mathcal{N}}))$

5. $G_{M_L} \leftarrow remove\_nodes(remove\_edges(G_M, \mathcal{E}_{G_{M_R}}), \mathcal{N} - \mathcal{N}_{G_{M_L}})$
6. $M_L \leftarrow (G_{M_L}, \varnothing)$
7. $G_{M_R} \leftarrow remove\_nodes(remove\_edges(G_M, \mathcal{E}_{G_{M_L}}), \mathcal{N}_{G_{M_L}} - \mathcal{N}_H)$
8. $M_R \leftarrow (G_{M_R}, \varnothing)$

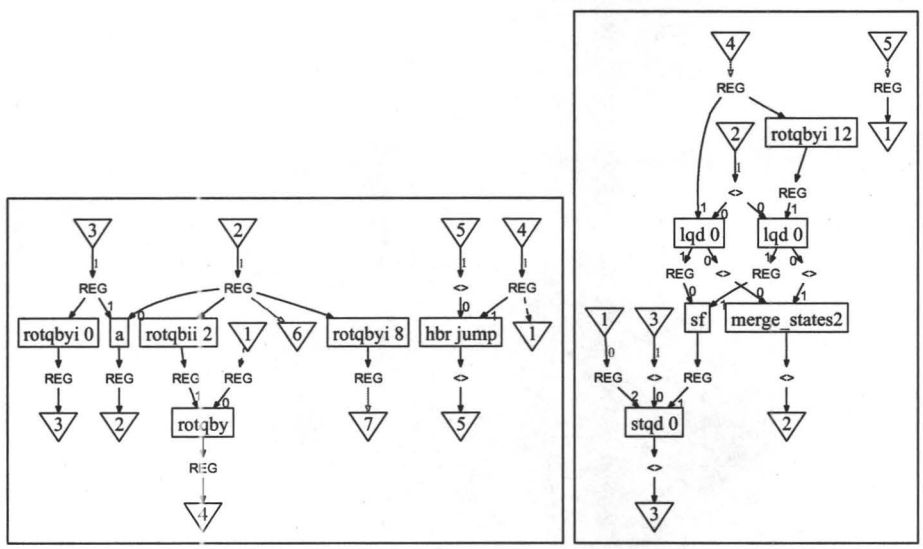
To demonstrate the above algorithm, it is applied to Figure $4.8(c)_{33}$. As argument to the algorithm, $\mathcal{E}_{G_{M_L}}$ is a singleton set containing only "rotqbyi 8", then $\mathcal{E}_{G_{M_L}}$ must contain the other edge "rotqbyi 12". The decomposition of edge set is possible to uniquely decompose $\mathsf{MapTicker}'_M$. Figure 4.10_{35} shows a final step of decomposing an extendable loop specification into two parts, it composes left middle to left and right middle to right,

$$X = (X_L \circledast X_{M_L}) \circledast (X_{M_R} \circledast X_R).$$



(a) $\mathsf{MapTicker}'_{M_L}$ (b) $\mathsf{MapTicker}'_{M_R}$

Figure 4.9: Decomposition of $\mathsf{MapTicker}'_M$



(a) $\mathsf{MapTicker}'_L \circledast \mathsf{MapTicker}'_{M_L}$

(b) $\mathsf{MapTicker}'_{M_R} \circledast \mathsf{MapTicker}'_R$

Figure 4.10: Final Decomposition of MapTicker′

Chapter 5

Relation Weighted CFG

A relation weighted control flow graph(RWCFG)[SS93],[SHW97] is a control flow graph weighted by heterogeneous relations. Pure control flow graphs are completely abstract over how state transitions are achieved; theories of pure control flow graphs are formulated in Kleene algebra. Since more information is given about states, the capability of exploring more theories increases. When all states are given in formal specifications, specification of all transitions is given for free; RWCFG is a relational approach, RWCFG relationally specifies all transition and only start and end states, specification of other states can be derived just like axiomatic semantics.

The two level nested control flow graphs in [AK08a] are very suitable to be modeled by RWCFG; since the outer level is a control flow graph, the inner level are relations, inner level graphs are edges of control flow graph. We extend the original RWCFG structure with types, which label nodes in a control flow graph; these labels match the types of relations between two nodes. In this chapter, two theorems Section 5.2_{39} and Section 5.3_{41} in RWCFGs are presented after an extended definition of RWCFGs Section 5.1_{37}.

5.1 Definition

Def. 5.1_{37} extends the original definition[SS93],[SHW97] with types; Def. 5.2_{38} shows how to relationally capture the behavior of RWCFG. Def. 5.3_{39} defines a convenient way to refer to a node in the flow graph of a RWCFG.

Definition 5.1 *A typed RWCFG is* $\mathcal{P} = (G, S, \Theta, e, a, \mathcal{T}, \tau)$*, provided*

- $S = (V, B)$ *is a graph, called the situation graph.*
- $G = (V_G, B_G)$ *is a graph, the underlying flowgraph.*

- $\Theta : V \rightarrow V_G$ *is a surjective graph homomorphism of* S *onto* G*; therefore,* Θ *is subject to:*

$$\mathbb{I} \subseteq \Theta;\Theta^{\smile},\ \Theta^{\smile};\Theta = \mathbb{I},\ B;\Theta \subseteq \Theta;B_G$$

- $\mathcal{T}$ *is a set of type objects*

- $\tau : V_G \rightarrow T$ *is a univalent typing relation. There is no type variable and polymorphism,* τ *is a function.*

- e *represents an input relation, satisfying,*

$$e^{\smile};e = \mathbb{I},\ e;e^{\smile} \subseteq \mathbb{I},\ e;\mathbb{T} = \Theta;\Theta^{\smile};e;\mathbb{T},\ \Theta^{\smile};e = \Theta^{\smile};e;\mathbb{T}$$

- a *represents an output relation, satisfying,*

$$a^{\smile};a = \mathbb{I},\ a;a^{\smile} \cap \Theta;\Theta^{\smile} \subseteq \mathbb{I},\ a;\mathbb{T} = \Theta;\Theta^{\smile};a;\mathbb{T}$$

Θ is a node mapping from the set of nodes in S to the set of nodes G. Moreover Θ is a surjective graph homomorphism to preserve the structure of S in G. The three conditions are formed by adding surjective condition $\mathbb{I} \subseteq \Theta^{\smile};\Theta$ to graph homomorphism conditions. $B;\Theta \subseteq \Theta;B_G$ has three other equivalent forms, $\Theta^{\smile};B;\Theta \subseteq B_G$, $\Theta^{\smile};B \subseteq B_G;\Theta^{\smile}$, $B \subseteq \Theta;B_G;\Theta^{\smile}$.

Def. 5.2_{38} captures the entire RWCFG as a relation; it can be used to argue behaviors between RWCFGs. It is originally defined in [SS93].

Definition 5.2 *Let* $\mathcal{P}$ *be a RWCFG, then* $\mathcal{P} = (G, S, \Theta, e, a, \mathcal{T}, \tau)$. $\Sigma(\mathcal{P})$ *is called effect of* $\mathcal{P}$*, such that*

$$\Sigma(\mathcal{P}) = e^{\smile};(B^* \cap \overline{B;\mathbb{T}}^{\smile});a$$

There is a minor difference between the RWCFG definitions in [SS93] and [SHW97]. [SHW97] dropped disjointness of e and a. This change allows a situation node to be related to both input and output, let us call such node *connector* if it is only connected by input and output. The consequence is that in some cases the associated relation B can be more concise than the one in [SS93]. If one wants to relate one element in input to one element in output, [SS93] requires a "dummy" edge. However, a connector can be easily forgotten in some analysis purely on B, such as reachability. In order to be aware of such connectors, in some proofs e and a have to be carried around. A self loop can be added to connectors to make the connector visible, but it is not forced by the definition. With disjointness of e and a, we can strengthen the definition of $\Sigma(\mathcal{P})$, i.e., $e^{\smile};(B^* \cap \overline{B;\mathbb{T}}^{\smile});a = e^{\smile};(B^+ \cap \overline{B;\mathbb{T}}^{\smile});a$

Proof:

$$
\begin{array}{lll}
 & e^{\smile};(B^{+} \cap \overline{B;\top}^{\smile});a & \\
\subseteq & e^{\smile};(B^{*} \cap \overline{B;\top}^{\smile});a & B^{+} \subseteq B^{*} \\
= & e^{\smile};(B^{+} \cap \overline{B;\top}^{\smile});a \cup e^{\smile};(\mathbb{I} \cap \overline{B;\top}^{\smile});a & \text{above} \\
\subseteq & e^{\smile};(B^{+} \cap \overline{B;\top}^{\smile});a \cup (e^{\smile};\mathbb{I};a \cap e^{\smile};\overline{B;\top}^{\smile};a) & \text{semi-distribution} \\
= & e^{\smile};(B^{+} \cap \overline{B;\top}^{\smile});a \cup (e^{\smile};a \cap e^{\smile};\overline{B;\top}^{\smile};a) & \text{identity of } ; \\
= & e^{\smile};(B^{+} \cap \overline{B;\top}^{\smile});a \cup (\bot\!\!\!\bot \cap e^{\smile};\overline{B;\top}^{\smile};a) & \text{Lemma A.9} \\
= & e^{\smile};(B^{+} \cap \overline{B;\top}^{\smile});a \cup \bot\!\!\!\bot & \\
= & e^{\smile};(B^{+} \cap \overline{B;\top}^{\smile});a & \text{identity of } \cup \qquad \square
\end{array}
$$

Definition 5.3 *Let $\mathcal{P} = (G, S, \Theta, e, a, \mathcal{T}, \tau)$ be a typed RWCFG and $S = (V, B)$, $G = (V_G, B_G)$, $\mathcal{N}_\mathcal{P}$ is a set of partial identity relations, each element represents a position in G. For every $x \in \mathcal{N}_\mathcal{P}$, x is subject to:*

$$
\begin{array}{rl}
x \neq \bot\!\!\!\bot & x \text{ is not empty,} \\
x \subseteq \mathsf{dom}\,(B) & x \text{ is a partial identity,} \\
x;(\Theta;\Theta^{\smile});x = x;\top;x & x \text{ focused on one class in the equivalence class } \Theta;\Theta^{\smile}, \\
x = \mathsf{ran}\,(x;\Theta;\Theta^{\smile}) & x \text{ covers at least one class in } \Theta;\Theta^{\smile}. \qquad \square
\end{array}
$$

For convenience, let $n_a \in \mathcal{N}_\mathcal{P}$ and $\mathsf{ran}\,(n_a;\Theta) = \{(a, a)\} \in V_G \times V_G$, n_a represents node a in V_G. n_e represents the initial node in B, it satisfies $n_e = \mathsf{dom}\, e$, likewise, there exists a $\mathcal{N}'_\mathcal{P}$, a proper subset of $\mathcal{N}_\mathcal{P}$, whose join represents accept nodes in B_G. Such that,

$$\mathcal{N}_a = \bigcup \mathcal{N}'_\mathcal{P} = \mathsf{dom}\, a, \text{ where } \mathcal{N}'_\mathcal{P} \subset \mathcal{N}_\mathcal{P}$$

Again n_e and $\mathcal{N}_a$ are disjoint, such that $n_e \cap \mathcal{N}_a = \bot\!\!\!\bot$.

5.2 Dead-Branch Introduction

In control flow graph, if an edge is never taken then removing or adding this edge does not change the behavior of the control flow graph. Usually never taken edge s appear as a branch of states; if a never taken edge is the only edge out going from a state then this state is a dead state. The next theorem, Theorem 5.1_{39}, formally proves the property of adding never-taken-edges, meanwhile a new RWCFG is constructed with the never-taken-edges. Also this theorem can be used as dead branch elimination by specifying $\mathcal{P}'$.

Theorem 5.1 (Never-Taken-Edge)
Let $\mathcal{P} = (G, S, \Theta, e, a, \mathcal{T}, \tau)$, $G = (V_G, B_G)$, $S = (V, B)$, $x, y \in V_G$. Let $\mathcal{P}' = (G', S', \Theta', e', a', \mathcal{T}, \tau)$, where $S' = (V, B \cup R)$, $G' = (V_G, B_G \cup \{(x, y)\})$, $\Theta' = \Theta$, $a' = a$, $e' = e$. With the following assumptions,

i. $x \neq e_G$.

ii. $\mathsf{ran}\,(B) \cap \mathsf{dom}\,R = \bot\!\!\!\bot$.

then $\Sigma(\mathcal{P}) = \Sigma(\mathcal{P}')$

Proof:
Assumption i implies,

$$\mathsf{ran}\,(e^{\smile}) \cap \mathsf{dom}\,R = \bot\!\!\!\bot$$

Also with Lemma A.9,

$$\mathsf{ran}\,(e^{\smile}) \cap \mathsf{dom}\,R = \bot\!\!\!\bot \Rightarrow e^{\smile};R = \bot\!\!\!\bot \tag{5.1}$$

Next we show $e^{\smile};((\mathcal{B} \cup R)^+ \cap \overline{(B \cup R);\mathbb{T}}^{\smile}) = e^{\smile};(B^+ \cap \overline{B;\mathbb{T}}^{\smile})$, also $a' = a$, then $\Sigma(\mathcal{P}) = \Sigma(\mathcal{P}')$:

$$\begin{array}{lll}
 & e^{\smile};((B \cup R)^+ \cap \overline{(B \cup R);\mathbb{T}}^{\smile}) & \\
= & e^{\smile};((B \cup R);(B \cup R)^* \cap \overline{(B \cup R);\mathbb{T}}^{\smile}) & \\
= & e^{\smile};(B;(B \cup R)^* \cup R;(B \cup R)^*) \cap \overline{(B \cup R);\mathbb{T}}^{\smile}) & \text{Distribution} \\
= & e^{\smile};((B^+ \cup R;(\mathcal{B} \cup R)^*) \cap \overline{(B \cup R);\mathbb{T}}^{\smile}) & \text{Lemma A.12} \\
= & e^{\smile};(B^+ \cap \overline{(B \cup R);\mathbb{T}}^{\smile} \cup R;(B \cup R)^* \cap \overline{(B \cup R);\mathbb{T}}^{\smile}) & \text{Distribution} \\
= & e^{\smile};(B^+ \cap \overline{(B \cup R);\mathbb{T}}^{\smile}) \cup e^{\smile};(R;(B \cup R)^* \cap \overline{(B \cup R);\mathbb{T}}^{\smile}) & \text{Distribution} \\
= & e^{\smile};(B^+ \cap \overline{(B \cup R);\mathbb{T}}^{\smile}) \cup \bot\!\!\!\bot & \text{Lemma A.13} \\
= & e^{\smile};(B^+ \cap \overline{(B \cup R);\mathbb{T}}^{\smile}) & \text{Identity of } \cup \\
= & e^{\smile};(B^+ \cap \overline{(B;\mathbb{T} \cup R;\mathbb{T})}^{\smile}) & \text{Distribution} \\
= & e^{\smile};(B^+ \cap (\overline{B;\mathbb{T}} \cap \overline{R;\mathbb{T}})^{\smile}) & \text{De Morgen} \\
= & e^{\smile};(B^+ \cap \overline{B;\mathbb{T}}^{\smile} \cap \overline{R;\mathbb{T}}^{\smile}) & \text{Distribution} \\
= & e^{\smile};(B^+ \cap \overline{B;\mathbb{T}}^{\smile}) & \text{below}
\end{array}$$

The following shows that with assumption ii we can prove $B^+ = B^+ \cap \overline{R;\mathbb{T}}^{\smile}$:

$$\begin{array}{lll}
 & \mathsf{ran}\,B \cap \mathsf{dom}\,R = \bot\!\!\!\bot & \\
\Leftrightarrow & \mathsf{ran}\,(\mathbb{I};B) \cap \mathsf{dom}\,(R;\mathbb{I}) = \bot\!\!\!\bot & \\
\Leftrightarrow & \mathsf{ran}\,(\mathsf{ran}\,(B^*);B) \cap \mathsf{dom}\,(R;\mathsf{dom}\,\mathbb{T}) = \bot\!\!\!\bot & \\
\Leftrightarrow & \mathsf{ran}\,(B^*;B) \cap \mathsf{dom}\,(R;\mathbb{T}) = \bot\!\!\!\bot & \text{Lemma A.6 and Lemma A.7} \\
\Leftrightarrow & \mathsf{ran}\,(B^+) \cap \mathsf{dom}\,(R;\mathbb{T}) = \bot\!\!\!\bot & \\
\Leftrightarrow & \mathsf{ran}\,(B^+) \cap \mathsf{ran}\,((R;\mathbb{T})^{\smile}) = \bot\!\!\!\bot & \\
\Rightarrow & B^+ \cap (R;\mathbb{T})^{\smile} = \bot\!\!\!\bot & \text{Lemma A.15} \\
\Rightarrow & B^+ \cap \overline{(R;\mathbb{T})^{\smile}} = B^+ & \text{Lemma A.8} \\
\Leftrightarrow & B^+ \cap \overline{(R;\mathbb{T})}^{\smile} = B^+ &
\end{array}$$

The last step is to show $\mathcal{P}'$ satisfies RWCFG definition. Since Θ', e', a' is equal to their counterparts in $\mathcal{P}$, the only not so trivial condition is homomorphism from S' to G'. B'_G can be also defined with point free style, $B'_G = B_G \cup \Theta^{\smile};n_x;\mathbb{T};n_y;\Theta$. Then the following is a proof of $B';\Theta' \subseteq \Theta';B'_G$:

$$
\begin{array}{lll}
 & B';\Theta' & \\
= & (B \cup R);\Theta' & \text{definition} \\
= & (B \cup R);\Theta & \Theta' = \Theta \\
= & B;\Theta \cup R;\Theta & \text{distribution} \\
\subseteq & \Theta;B_G \cup R;\Theta & \mathcal{P} \text{ is RWCFG and monotonicity of } \cup \\
= & \Theta;B_G \cup n_x;R;n_y;\Theta & \text{condition of } R \\
= & \Theta;B_G \cup \mathbb{I};n_x;R;n_y;\Theta & \text{identity of } ; \\
\subseteq & \Theta;B_G \cup \Theta;\Theta^{\smile};n_x;\mathbb{T};n_y;\Theta & \text{monotonicity of } \cup \text{ and } ;,\ \mathbb{I} \subseteq \Theta;\Theta^{\smile},\ R \subseteq \mathbb{T} \\
= & \Theta;(B_G \cup \Theta^{\smile};n_x;\mathbb{T};n_y;\Theta) & \text{distribution} \\
= & \Theta;(B_G \cup \Theta^{\smile};n_x;\mathbb{T};n_y;\Theta) & \Theta' = \Theta \\
= & \Theta';B'_G & \square
\end{array}
$$

Theorem 5.1_{39} is inspired by a graph transformation in [AK08a, Page 28]. RWCFG provides more information of specification of state transfer, whereas in CFG such computation is abstractly represented by edge label. With this additional knowledge, a never taken edge can be expressed in RWCFG, but impossible in CFG. Moreover, inserting a never taken edge does not change the semantics in RWCFG as shown in Theorem 5.1_{39}. A useful application of a never taken edge is documented in [AK08a]. Adding such "dummy" edge in a RWCFG may have more opportunities to be simplified using standard control flow graph minimization algorithm. Figure 5.1 is a simple example that shows adding a never-taken-edge can simplify RWCFG. For simplicity only flow graphs of RWCFGs are drawn. The left graph is original RWCFG. Given the sufficient condition $\mathsf{ran}\ P \cap \mathsf{dom}\ R = \bot\!\!\!\bot$, the red dash edge inserted from the right node in the second row to left node in the third row in the middle graph is a never-taken-edge. Theorem 5.1_{39} guarantees the left and middle graph has the same semantics. Then the right graph is the result after applying CFG minimization algorithm on the middle graph. Edge symbols in RWCFG are relations, if two edges share the same source and target nodes then they can be unioned into one edge or relation.

5.3 Edge Replacement

The next theorem can be used to formally justify sequential edge decomposition; sequential edge decomposition is presented in [AK08a]. In [AK08a], some

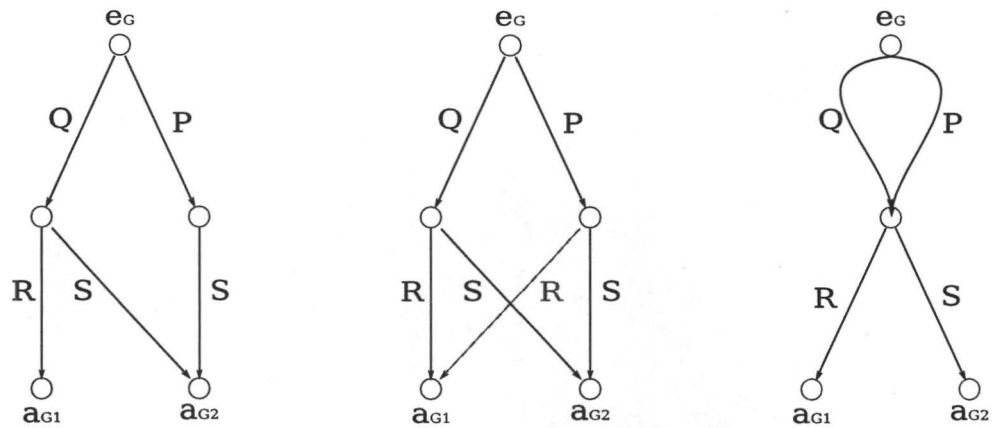


Figure 5.1: Application of Never-Taken-Edge

edges in control flow graph are decomposed into three parts; then after rearranging edges, better instruction scheduling can be achieved. Decomposing an edge in a control flow graph intuitively looks correct, in our view, decomposing an edge is the same as replacing the old edge by another control flow graph. The next theorem gives a formal way to construct a new RWCFG by replacing an edge with another RWCFG; also the next theorem helps to show that an edge decomposition is a closed operator.

Theorem 5.2 (Edge Replacement)
Given $\mathcal{P}_1 = (G_1, S_1, \Theta_1, e_1, a_1, \mathcal{T}_1, \tau_1)$ *and* $\mathcal{P}_2 = (G_2, S_2, \Theta_2, e_2, a_2, \mathcal{T}_2, \tau_2)$. *In* $\mathcal{P}_1$ *replacing edge between node* x *and node* y *by* $\mathcal{P}_2$ *with two interface relations* μ *and* ν *is again a RWCFG, denoted as* $\mathcal{P}_{1y}^{x}(\mathcal{P}_2)$. μ *is between node* x *and input of* $\mathcal{P}_2$ *and* ν *is between node* y *and output of* $\mathcal{P}_2$*; formally the conditions can be written as,*

$$\mathsf{dom}\ \mu \subseteq n_x,\ \ \mathsf{dom}\ \nu \subseteq n_y,\ \ \mathsf{ran}\ \mu \subseteq \mathsf{ran}\ e_2,\ \ \mathsf{ran}\ \nu \subseteq \mathsf{ran}\ a_2$$

Proof:

Let	$V = V_1 \uplus V_2,$	$V_G = V_{G_1} \uplus V_{G_2},$
	$e_2 : V_2 \leftrightarrow \mathcal{A},$	$a_2 : V_2 \leftrightarrow \mathcal{A},$
then	$\mu : V_1 \leftrightarrow \mathcal{A},$	$\nu : V_1 \leftrightarrow \mathcal{A};$
recall Definition A.2:	$\iota_V : V_1 \rightarrow V,$	$\kappa_V : V_2 \rightarrow V,$
	$\iota_{V_G} : V_{G_1} \rightarrow V_G,$	$\kappa_{V_G} : V_{G_2} \rightarrow V_G.$

The elements in $\mathcal{P}_{1y}^{x}(\mathcal{P}_2)$ are

- $S = (V, \alpha_1 \cup \alpha_2 \cup \alpha_3)$
  - $\alpha_1 = (B_1 \cap \overline{n_x ; \mathbb{T} ; n_y}) \uplus B_2$
  - $\alpha_2 = \iota_V^{\smile} ; \mu ; e_2^{\smile} ; \kappa_V$
  - $\alpha_3 = \kappa_V^{\smile} ; a_2 ; \nu^{\smile} ; \iota_V)$
- $G = (V_G, \beta_1 \cup \beta_2 \cup \beta_3)$
  - $\beta_1 = (B_{G_1} \cap \overline{\Theta_1^{\smile} ; n_x ; \mathbb{T} ; n_y ; \Theta_1}) \uplus B_{G_2}$
  - $\beta_2 = \iota_{V_G}^{\smile} ; \Theta_1^{\smile} ; n_x ; \mathbb{T} ; n_{e_2} ; \Theta_2 ; \kappa_{V_G}$
  - $\beta_3 = \kappa_{V_G}^{\smile} ; \Theta_2^{\smile} ; n_{a_2} ; \mathbb{T} ; n_y ; \Theta_1 ; \iota_{V_G}$
- $\Theta = \Theta_1 \uplus \Theta_2 = \iota_V^{\smile} ; \Theta_1 ; \iota_{V_G} \cup \kappa_V^{\smile} ; \Theta_2 ; \kappa_{V_G}$
- $e = \iota_V^{\smile} ; e_1$
- $a = \iota_V^{\smile} ; a_1$
- $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$
- $\tau = \iota_{V_G}^{\smile} ; \tau_1 \cup \kappa_{V_G}^{\smile} ; \tau_2$

The next step is to show $B_1 ; \Theta_1 \subseteq \Theta_1 ; B_{G_1}$, Proof 1_{55} and Proof 2_{56} expand $B_1 ; \Theta_1$ and $\Theta_1 ; B_{G_1}$ respectively. Both of them are expanded to a form of joining of four terms. Proof 3_{56}, Proof 4_{57}, Proof 5_{57},Proof 6_{57} show that each term from $B_1 ; \Theta_1$ is less than or equal to their counterpart in $\Theta_1 ; B_{G_1}$. Therefore, $B_1 ; \Theta_1 \subseteq \Theta_1 ; B_{G_1}$ is proven with monotonicity of $\cup$. □

Chapter 6

Conclusion and Future Work

It is personally believed that to achieve the goal of COCONUT it is a good idea to always think in formal methods. Although applying formal methods is time consuming and expensive, but it is worth, at least implicitly, to develop such meta-development framework. We hope that the work of this thesis could be some help to COCONUT for moving forward in the rigorous development.

Extensible loop specification can only compose from a list or decompose to a list, a more general or desired structure is directed acyclic graph, or term graph, which needs to define $\otimes$ and necessary constants in extensible loop specification.

It is possible to implement theories of RWCFG in a theorem prover; it would be more efficient to prove properties of two level nested graphs like in [AK08a].

Appendices

Appendix A

Relation Algebra

A relation is a generalization of a function. Functions are deterministic whereas relations are, in general, nondeterministic. For example, if f is a function from natural numbers to natural numbers, $f : \mathbb{N} \to \mathbb{N}$, for each input to f, there is a unique output. A relation from natural numbers to natural numbers $R : \mathbb{N} \leftrightarrow \mathbb{N}$, which is a subset of $\mathbb{N} \times \mathbb{N}$. If (x, y) is in relation R, we write xRy. If xRy and xRz, it does not follow that $y = z$. In this chapter a minimum set of topics are presented to support later chapters, mainly Chapter 5_{37}. Section A.1_{49} introduces the two primary operators composition $\fatsemi$ and converse $\smile$, which are not defined in set theory. Section A.2_{51} introduces direct sum of relations in an abstract way such that it is described by axioms which only involve relation operators and equality. Section A.3_{52} includes some lemmas which are used in the later proofs.

A.1 Definition and Operators

Formally, a relation is a set of pairs R of $X \times Y$,

$$R \subseteq X \times Y$$

If $X = Y$, then R is called a homogeneous relation, otherwise it is called a heterogeneous relation. The empty relation is an empty set, denoted as $\bot\!\!\!\bot$, the universal relation is $X \times Y$, denoted as $\top$, the identity relation is $\{(a, a) \mid a \in X\}$, denoted as $\mathbb{I}_X$, the identity relation must be homogeneous. Operators defined on relations can be divided into two groups, operators from set theory and relation operators. Set operators are union ($\cup$), intersect ($\cap$) and complement ($\overline{\ }$). All theorems in set theory are inherited as well. Lemma A.8 is proved in set theory. Relation operators are composition ($\fatsemi$) and converses ($\breve{\ }$), defined as,

composition $x(X;Y)y$ iff there exits z such that xXz and zYy. Derived properties are,

unit: $\mathbb{I};X = X = X;\mathbb{I}$

associativity: $(X;Y);Z = X;(Y;Z)$

distributivity over $\cup$**:** $X;(Y \cup Z) = (X;Y) \cup (X;Z)$ and $(Y \cup Z);X = (Y;X) \cup (Z;X)$

semi-distributivity over $\cap$**:** $X;(Y \cap Z) \subseteq (X;Y) \cap (X;Z)$ and $(Y \cup Z);X \subseteq (Y;X) \cap (Z;X)$

converse $xX^{\smile}y$ iff yXx. Derived properties are,

involution: $X^{\smile\smile} = X$

distributivity over $\cup$**:** $(X \cup Y)^{\smile} = X^{\smile} \cup Y^{\smile}$

distributivity over $\cap$**:** $(X \cap Y)^{\smile} = X^{\smile} \cap Y^{\smile}$

anti-distributivity over $;$**:** $(X;Y)^{\smile} = Y^{\smile};X^{\smile}$

distributivity over $\overline{\ }$**:** $\overline{X}^{\smile} = \overline{X^{\smile}}$

$\mathsf{dom}\,(R)$ and $\mathsf{ran}\,(R)$ capture the domain and the range of relation R; $\mathsf{dom}\,(R)$ and $\mathsf{ran}\,(R)$ are two subidentities containing only elements in domain and range of R respectively; they are defined based on existing operators,

$$\mathsf{dom}\,(R) = \mathbb{I} \cap (R;R^{\smile}),\ \mathsf{ran}\,(R) = \mathbb{I} \cap (R^{\smile};R)$$

Definition A.1 *Transitive closure of relation R, written as R^+, is defined as following,*

$$R^+ = \{i : \mathbb{N} \mid i \geqslant 1 \bullet R^i\}$$

Reflexive and transitive closure is then defined as $R^* = \mathbb{I} \cup R^+$.

Lemma A.1 (Monotonicity of $;$ and $\cup$) *Given $X \subseteq W$ and $Y \subseteq Z$, one has*

$$X;Y \subseteq W;Z,\ X \cup Y \subseteq W \cup Z\ .$$

Theorem A.1 (Dedekind Formula)
Given Q, R, and S, one has

$$Q;R \cap S \subseteq (Q \cap S;R^{\smile});(R \cap Q^{\smile};S)$$

Theorem A.2
Given relations A, B, and C, one has

$$A;B \subseteq C \Leftrightarrow B;\overline{C^{\smile}} \subseteq \overline{A^{\smile}}$$

Theorem A.2_{51} is an equivalent form to Schröder equivalences; it is easier to apply and memorize than Schröder equivalences. "$\Rightarrow$" cyclically rotates to left, "$\Leftarrow$" rotates to right. In each direction, two literals are conversed and complemented.

A.2 Direct Sum

[Kah06] and [Kah01] provide a relational specification of direct sum in an abstract way, four relation equations capture the definition of direct sum of two objects, which is a base of definition of direct sum of relations.

Definition A.2 *A direct sum of two objects $\mathcal{A}$ and $\mathcal{B}$ is a triple $(\mathcal{S}, \iota, \kappa)$, $\mathcal{S}$ is an object and ι, κ are two injective mappings and the following conditions hold:*

$$\iota;\iota^{\smile} = \mathbb{I}_{\mathcal{A}},\ \kappa;\kappa^{\smile} = \mathbb{I}_{\mathcal{B}},\ \iota;\kappa^{\smile} = \perp\!\!\!\perp_{\mathcal{A},\mathcal{B}},\ \iota^{\smile};\iota \cap \kappa^{\smile};\kappa = \mathbb{I}_{\mathcal{S}}.$$

□

Let operator $\uplus$ be one way to construct direct sum and subscript ι and κ with direct summed object. For example, in the above case $\mathcal{S} = \mathcal{A} \uplus \mathcal{B}$ with $\iota_{\mathcal{S}}$ and $\kappa_{\mathcal{S}}$. Another convention is that $\iota_{\mathcal{S}}$ is an injection for the left operand of $\uplus$ whereas $\kappa_{\mathcal{S}}$ is for the right one. Then the signature of $\iota_{\mathcal{S}}$ and $\kappa_{\mathcal{S}}$ are $\mathcal{A} \rightarrow \mathcal{S}$ and $\mathcal{B} \rightarrow \mathcal{S}$ respectively. $\uplus$ is not communitive, since commuting two operands requires changing the role of ι and κ. $\uplus$ is not associative, for instance $\mathcal{S} = (\mathcal{A} \uplus \mathcal{B}) \uplus \mathcal{C}$, $\mathcal{T} = \mathcal{A} \uplus (\mathcal{B} \uplus \mathcal{C})$, $\iota_{\mathcal{S}} \neq \iota_{\mathcal{T}}$.

The definition of direct sum of two relations is taken from [Kah06].

Definition A.3 *Given relation $X : \mathcal{A} \leftrightarrow \mathcal{B}$ and $Y : \mathcal{C} \leftrightarrow \mathcal{D}$, let $\mathcal{S} = \mathcal{A} \uplus \mathcal{C}$ and $\mathcal{T} = \mathcal{B} \uplus \mathcal{D}$. Direct sum of X and Y is denoted as $X \uplus Y$ with four injections $\iota_{\mathcal{S}}$,$\iota_{\mathcal{T}}$,$\kappa_{\mathcal{S}}$,$\kappa_{\mathcal{T}}$,*

$$X \uplus Y = \iota_{\mathcal{S}}^{\smile};X;\iota_{\mathcal{T}} \cup \kappa_{\mathcal{S}}^{\smile};Y;\kappa_{\mathcal{T}}$$

Lemma A.2 *Given relation $X : \mathcal{A} \leftrightarrow \mathcal{B}$ and $Y : \mathcal{C} \leftrightarrow \mathcal{D}$, let $Z = X \uplus Y$, $\mathcal{S} = \mathcal{A} \uplus \mathcal{C}$ and $\mathcal{T} = \mathcal{B} \uplus \mathcal{D}$; then $\iota_{\mathcal{S}};Z;\kappa_{\mathcal{T}}^{\smile} = \perp\!\!\!\perp$*

Proof:

$$\begin{array}{lll}
\iota_{\mathcal{S}};Z;\kappa_{\mathcal{T}}^{\smile} & = \iota_{\mathcal{S}};(\iota_{\mathcal{S}}^{\smile};X;\iota_{\mathcal{T}} \cup \kappa_{\mathcal{S}}^{\smile};Y;\kappa_{\mathcal{T}});\kappa_{\mathcal{T}}^{\smile} & \text{Definition A.3} \\
& = \iota_{\mathcal{S}};\iota_{\mathcal{S}}^{\smile};X;\iota_{\mathcal{T}};\kappa_{\mathcal{T}}^{\smile} \cup \iota_{\mathcal{S}};\kappa_{\mathcal{S}}^{\smile};Y;\kappa_{\mathcal{T}};\kappa_{\mathcal{T}}^{\smile} & \text{Distribution} \\
& = \iota_{\mathcal{S}};\iota_{\mathcal{S}}^{\smile};X;\bot\!\!\!\bot \cup \bot\!\!\!\bot;Y;\kappa_{\mathcal{T}};\kappa_{\mathcal{T}}^{\smile} & \text{Definition A.2} \\
& = \bot\!\!\!\bot \cup \bot\!\!\!\bot & \text{Zero laws} \\
& = \bot\!\!\!\bot &
\end{array}$$

A.3 Lemmas

A set of lemmas is presented, all of them are used in later chapters; some of them are taken from other sources, some are proven immediately.

Lemma A.3 $\mathsf{dom}\, X \cap \mathsf{dom}\, Y = \mathsf{dom}\, X;\mathsf{dom}\, Y$[Kah01]

Lemma A.4 $X = \mathsf{dom}\, X;X$[Kah01]

Lemma A.5 $X = X;\mathsf{ran}\, X$[Kah01]

Lemma A.6 $\mathsf{ran}\,(X;Y) = \mathsf{ran}\,(\mathsf{ran}\,(X);Y)$[Kah01]

Lemma A.7 $\mathsf{dom}\,(X;Y) = \mathsf{ran}\,(X;\mathsf{dom}\, Y)$[Kah01]

Lemma A.8 $X \cap Y = \bot\!\!\!\bot \Rightarrow X \subseteq \overline{Y}$

Proof:

$$\begin{array}{lll}
X \cap \overline{Y} & = (X \cap \overline{\overline{Y}}) \cup \bot\!\!\!\bot & \text{Identity of } \cup \\
& = (X \cap \overline{\overline{Y}}) \cup (X \cap Y) & \text{Assumption} \\
& = X \cap (\overline{\overline{Y}} \cup Y) & \text{Distribution of } \cap \text{ over } \cup \\
& = X \cap \top\!\!\!\top & \\
& = X & X \subseteq \top\!\!\!\top \\
\Rightarrow & X \subseteq \overline{Y} & \\
\Leftrightarrow & X \cap \overline{Y} = X &
\end{array}$$

Lemma A.9 $\mathsf{ran}\, X \cap \mathsf{dom}\, Y = \bot\!\!\!\bot \Rightarrow X;Y = \bot\!\!\!\bot$

Proof:

$$\begin{array}{lll}
X;Y & = (X;\mathsf{ran}\, X);(\mathsf{dom}\, Y;Y) & \text{Lemma A.4, Lemma A.5} \\
& = X;(\mathsf{ran}\, X;\mathsf{dom}\, Y);Y & \text{Associativity} \\
& = X;(\mathsf{ran}\, X \cap \mathsf{dom}\, Y);Y & \text{Lemma A.3} \\
& = X;\bot\!\!\!\bot;Y & \text{Assumption} \\
& = \bot\!\!\!\bot &
\end{array}$$

Lemma A.10 $\mathsf{ran}\, X \cap \mathsf{dom}\, Y = \bot\!\!\!\bot \Rightarrow X^*;Y = Y$

Proof:

$$
\begin{array}{lll}
X^*;Y & = (\mathbb{I} \cup X^*;X);Y & \text{[Koz94] Proposition 2} \\
& = (\mathbb{I};Y) \cup (X^*;X;Y) & \text{Distribution} \\
& = Y \cup (X^*;(X;Y)) & \text{Identity of ;, associativity} \\
& = Y \cup (X^*;\perp\!\!\!\perp) & \text{Lemma A.9} \\
& = Y \cup \perp\!\!\!\perp & \\
& = Y & \text{Identity of } \cup
\end{array}
$$

Lemma A.11 $\mathsf{ran}\, X \cap \mathsf{dom}\, Y = \perp\!\!\!\perp \Rightarrow X;Y^* = X$

Proof:

$$
\begin{array}{lll}
X;Y^* & = (X;Y^*)^{\smile} & \\
& = (Y^{*\smile};X^{\smile})^{\smile} & \\
& = (Y^{\smile *};X^{\smile})^{\smile} & \\
& = X^{\smile\smile} & \mathsf{ran}\,(R^{\smile}) = \mathsf{dom}\, R \text{ and Lemma A.10} \\
& = X &
\end{array}
$$

Lemma A.12 $\mathsf{ran}\, X \cap \mathsf{dom}\, Y = \perp\!\!\!\perp \Rightarrow X;(X \cup Y)^* = X^+$

Proof:

$$
\begin{array}{lll}
X;(X \cup Y)^* & = X;Y^*;(X;Y^*)^* & \text{[Koz94] Proposition 7} \\
& = X;Y^*;X^* & \text{Lemma A.11} \\
& = X;X^* & \text{Lemma A.11} \\
& = X^+ &
\end{array}
$$

Lemma A.13 $\mathsf{ran}\, X \cap \mathsf{dom}\, Y = \perp\!\!\!\perp \Rightarrow X;(Y;W \cap Z) = \perp\!\!\!\perp$

Proof:

$$
\begin{array}{lll}
\perp\!\!\!\perp & \subseteq X;(Y;W \cap Z) & \\
& \subseteq X;Y;W \cap X;Z & \text{Distribution of ; over } \cap \\
& = \perp\!\!\!\perp;W \cap X;Z & \text{Lemma A.9} \\
& = \perp\!\!\!\perp \cap X;Z & \perp\!\!\!\perp;R = \perp\!\!\!\perp \\
& = \perp\!\!\!\perp & \perp\!\!\!\perp \cap R = \perp\!\!\!\perp
\end{array}
$$

Lemma A.14 $\mathsf{dom}\, X \cap \mathsf{dom}\, Y = \perp\!\!\!\perp \Rightarrow X \cap Y = \perp\!\!\!\perp$

Proof:

$$
\begin{array}{lll}
\perp\!\!\!\perp & \subseteq X \cap Y & \\
& = \mathsf{dom}\, X;X \cap \mathsf{dom}\, Y;Y & \text{Lemma A.4} \\
& \subseteq (\mathsf{dom}\, X \cap \mathsf{dom}\, Y;Y;X^{\smile});(X \cap (\mathsf{dom}\, X)^{\smile};\mathsf{dom}\, Y;Y) & \text{Dedekind Formula} \\
& = (\mathsf{dom}\, X \cap \mathsf{dom}\, Y;Y;X^{\smile});(X \cap \mathsf{dom}\, X;\mathsf{dom}\, Y;Y) & \\
& = (\mathsf{dom}\, X \cap \mathsf{dom}\, Y;Y;X^{\smile});(X \cap \mathsf{dom}\, X \cap \mathsf{dom}\, Y;Y) & \text{Lemma A.3} \\
& = (\mathsf{dom}\, X \cap \mathsf{dom}\, Y;Y;X^{\smile});(X \cap \perp\!\!\!\perp;Y) & \text{Assumption} \\
& = (\mathsf{dom}\, X \cap \mathsf{dom}\, Y;Y;X^{\smile});(X \cap \perp\!\!\!\perp) & \\
& = (\mathsf{dom}\, X \cap \mathsf{dom}\, Y;Y;X^{\smile});\perp\!\!\!\perp & \\
& = \perp\!\!\!\perp &
\end{array}
$$

Lemma A.15 $\mathsf{ran}\, X \cap \mathsf{ran}\, Y = \perp\!\!\!\perp \Rightarrow X \cap Y = \perp\!\!\!\perp$

Proof:

$$\begin{aligned}
\mathsf{ran}\, X \cap \mathsf{ran}\, Y = \perp\!\!\!\perp \quad & \Leftrightarrow \mathsf{dom}\,(X^{\smile}) \cap \mathsf{dom}\,(Y^{\smile}) = \perp\!\!\!\perp \\
& \Rightarrow X^{\smile} \cap Y^{\smile} = \perp\!\!\!\perp \\
& \Leftrightarrow (X \cap Y)^{\smile} = \perp\!\!\!\perp \\
& \Leftrightarrow (X \cap Y)^{\smile\smile} = \perp\!\!\!\perp^{\smile} \\
& \Leftrightarrow X \cap Y = \perp\!\!\!\perp
\end{aligned}$$

Appendix B

Proof

A collection of proofs are included here. All of them are needed in Chapter 5_{37},Theorem 5.2_{42}.

Proof 1

$$
\begin{array}{lll}
& B;\Theta & \\
= & ((B_1 \cap \overline{n_x;\mathbb{T};n_y}) \uplus B_2 \cup \iota_V^{\smile};\mu;e_2^{\smile};\kappa_V \cup \kappa_V^{\smile};a_2;\nu^{\smile};\iota_V) & \\
& ;(\iota_V^{\smile};\Theta_1;\iota_{V_G} \cup \kappa_V^{\smile};\Theta_2;\kappa_{V_G}) & \text{Def. } B,\ \Theta \\
= & ((B_1 \cap \overline{n_x;\mathbb{T};n_y}) \uplus B_2);(\iota_V^{\smile};\Theta_1;\iota_{V_G} \cup \kappa_V^{\smile};\Theta_2;\kappa_{V_G}) & \\
& \cup\ \iota_V^{\smile};\mu;e_2^{\smile};\kappa_V;\kappa_V^{\smile};\Theta_2;\kappa_{V_G} \cup \kappa_V^{\smile};a_2;\nu^{\smile};\iota_V;\iota_V^{\smile};\Theta_1;\iota_{V_G} & \\
& \cup\ \iota_V^{\smile};\mu;e_2^{\smile};\kappa_V;\iota_V^{\smile};\Theta_1;\iota_{V_G} \cup \kappa_V^{\smile};a_2;\nu^{\smile};\iota_V;\kappa_V^{\smile};\Theta_2;\kappa_{V_G} & \text{Distribution} \\
= & ((B_1 \cap \overline{n_x;\mathbb{T};n_y}) \uplus B_2);(\iota_V^{\smile};\Theta_1;\iota_{V_G} \cup \kappa_V^{\smile};\Theta_2;\kappa_{V_G}) & \\
& \cup\ \iota_V^{\smile};\mu;e_2^{\smile};\kappa_V;\kappa_V^{\smile};\Theta_2;\kappa_{V_G} \cup \kappa_V^{\smile};a_2;\nu^{\smile};\iota_V;\iota_V^{\smile};\Theta_1;\iota_{V_G} & \\
& \cup\ \iota_V^{\smile};\mu;e_2^{\smile};\perp\!\!\!\perp;\Theta_1;\iota_{V_G} \cup \kappa_V^{\smile};a_2;\nu^{\smile};\perp\!\!\!\perp;\Theta_2;\kappa_{V_G} & \text{Def. A.2 } \iota;\kappa^{\smile} = \perp\!\!\!\perp = \kappa;\iota^{\smile} \\
= & ((B_1 \cap \overline{n_x;\mathbb{T};n_y}) \uplus B_2);(\iota_V^{\smile};\Theta_1;\iota_{V_G} \cup \kappa_V^{\smile};\Theta_2;\kappa_{V_G}) & \\
& \cup\ \iota_V^{\smile};\mu;e_2^{\smile};\kappa_V;\kappa_V^{\smile};\Theta_2;\kappa_{V_G} \cup \kappa_V^{\smile};a_2;\nu^{\smile};\iota_V;\iota_V^{\smile};\Theta_1;\iota_{V_G} & \text{Property of } \perp\!\!\!\perp \\
= & ((B_1 \cap \overline{n_x;\mathbb{T};n_y}) \uplus B_2);(\iota_V^{\smile};\Theta_1;\iota_{V_G} \cup \kappa_V^{\smile};\Theta_2;\kappa_{V_G}) & \\
& \cup\ \iota_V^{\smile};\mu;e_2^{\smile};\mathbb{I}_{V_2};\Theta_2;\kappa_{V_G} \cup \kappa_V^{\smile};a_2;\nu^{\smile};\mathbb{I}_{V_1};\Theta_1;\iota_{V_G} & \text{Def. A.2 } \iota;\iota^{\smile} = \mathbb{I},\ \kappa;\kappa^{\smile} = \mathbb{I} \\
= & \iota_V^{\smile};\mu;e_2^{\smile};\Theta_2;\kappa_{V_G} \cup \kappa_V^{\smile};a_2;\nu^{\smile};\Theta_1;\iota_{V_G} & \\
& \cup\ ((B_1 \cap \overline{n_x;\mathbb{T};n_y}) \uplus B_2);(\iota_V^{\smile};\Theta_1;\iota_{V_G} \cup \kappa_V^{\smile};\Theta_2;\kappa_{V_G}) & \text{Commu. of } \cup,\ \text{Identity of } ; \\
= & \iota_V^{\smile};\mu;e_2^{\smile};\Theta_2;\kappa_{V_G} \cup \kappa_V^{\smile};a_2;\nu^{\smile};\Theta_1;\iota_{V_G} & \\
& \cup\ ((B_1 \cap \overline{n_x;\mathbb{T};n_y}) \uplus B_2);\iota_V^{\smile};\Theta_1;\iota_{V_G} & \\
& \cup\ ((B_1 \cap \overline{n_x;\mathbb{T};n_y}) \uplus B_2);\kappa_V^{\smile};\Theta_2;\kappa_{V_G} & \text{Distribution} \\
= & \iota_V^{\smile};\mu;e_2^{\smile};\Theta_2;\kappa_{V_G} \cup \kappa_V^{\smile};a_2;\nu^{\smile};\Theta_1;\iota_{V_G} & \\
& \cup\ (\iota_V^{\smile}(B_1 \cap \overline{n_x;\mathbb{T};n_y});\iota_V \cup \kappa_V^{\smile};B_2;\kappa_V);\iota_V^{\smile};\Theta_1;\iota_{V_G} & \\
& \cup\ (\iota_V^{\smile}(B_1 \cap \overline{n_x;\mathbb{T};n_y});\iota_V \cup \kappa_V^{\smile};B_2;\kappa_V);\kappa_V^{\smile};\Theta_2;\kappa_{V_G} & \text{Def. A.3} \\
= & \iota_V^{\smile};\mu;e_2^{\smile};\Theta_2;\kappa_{V_G} \cup \kappa_V^{\smile};a_2;\nu^{\smile};\Theta_1;\iota_{V_G} & \\
& \cup\ \iota_V^{\smile}(B_1 \cap \overline{n_x;\mathbb{T};n_y});\Theta_1;\iota_{V_G} \cup \kappa_V^{\smile};B_2;\Theta_2;\kappa_{V_G} & \text{Same as above}
\end{array}
$$

Proof 2

$$
\begin{aligned}
& \Theta;B_G \\
= \;& \Theta;((B_{G_1} \cap \overline{\breve{\Theta_1};n_x;\mathbb{T};n_y;\Theta_1}) \uplus B_{G_2}) \\
& \cup \Theta;(\breve{\iota_{V_G}};\Theta_1;n_x;\mathbb{T};n_{e_2};\Theta_2;\kappa_{V_G}) \\
& \cup \Theta;(\breve{\kappa_{V_G}};\Theta_2;n_{a_2};\mathbb{T};n_y;\Theta_1;\iota_{V_G}) && \text{Expand } B_G \\
= \;& \Theta;((B_{G_1} \cap \breve{\Theta_1};n_x;\mathbb{T};n_y;\Theta_1) \uplus B_{G_2}) \\
& \cup (\breve{\iota_V};\Theta_1;\iota_{V_G} \cup \breve{\kappa_V};\Theta_2;\kappa_{V_G});(\breve{\iota_{V_G}};\Theta_1;n_x;\mathbb{T};n_{e_2};\Theta_2;\kappa_{V_G}) \\
& \cup (\breve{\iota_V};\Theta_1;\iota_{V_G} \cup \breve{\kappa_V};\Theta_2;\kappa_{V_G});(\breve{\kappa_{V_G}};\Theta_2;n_{a_2};\mathbb{T};n_y;\Theta_1;\iota_{V_G}) && \text{Expand } \Theta \\
= \;& \Theta;((B_{G_1} \cap \breve{\Theta_1};n_x;\mathbb{T};n_y;\Theta_1) \uplus B_{G_2}) \\
& \cup \breve{\iota_V};\Theta_1;\iota_{V_G};\breve{\iota_{V_G}};\breve{\Theta_1};n_x;\mathbb{T};n_{e_2};\Theta_2;\kappa_{V_G} \\
& \cup \breve{\kappa_V};\Theta_2;\kappa_{V_G};\breve{\iota_{V_G}};\breve{\Theta_1};n_x;\mathbb{T};n_{e_2};\Theta_2;\kappa_{V_G} \\
& \cup \breve{\iota_V};\Theta_1;\iota_{V_G};\breve{\kappa_{V_G}};\breve{\Theta_2};n_{a_2};\mathbb{T};n_y;\Theta_1;\iota_{V_G} \\
& \cup \breve{\kappa_V};\Theta_2;\kappa_{V_G};\breve{\kappa_{V_G}};\breve{\Theta_2};n_{a_2};\mathbb{T};n_y;\Theta_1;\iota_{V_G} && \text{Distribution} \\
= \;& \Theta;((B_{G_1} \cap \breve{\Theta_1};n_x;\mathbb{T};n_y;\Theta_1) \uplus B_{G_2}) \\
& \cup \breve{\iota_V};\Theta_1;\mathbb{I}_{V_1};\breve{\Theta_1};n_x;\mathbb{T};n_{e_2};\Theta_2;\kappa_{V_G} && \text{Def. A.2} \\
& \cup \breve{\kappa_V};\Theta_2;\perp;\breve{\Theta_1};n_x;\mathbb{T};n_{e_2};\Theta_2;\kappa_{V_G} && \text{Def. A.2} \\
& \cup \breve{\iota_V};\Theta_1;\perp;\breve{\Theta_2};n_{a_2};\mathbb{T};n_y;\Theta_1;\iota_{V_G} && \text{Def. A.2} \\
& \cup \breve{\kappa_V};\Theta_2;\mathbb{I}_{V_2};\breve{\Theta_2};n_{a_2};\mathbb{T};n_y;\Theta_1;\iota_{V_G} && \text{Def. A.2} \\
= \;& \Theta;((B_{G_1} \cap \breve{\Theta_1};n_x;\mathbb{T};n_y;\Theta_1) \uplus B_{G_2}) \\
& \cup \breve{\iota_V};\Theta_1;\breve{\Theta_1};n_x;\mathbb{T};n_{e_2};\Theta_2;\kappa_{V_G} \cup \breve{\kappa_V};\Theta_2;\breve{\Theta_2};n_{a_2};\mathbb{T};n_y;\Theta_1;\iota_{V_G} && \text{Simplification} \\
= \;& \Theta;(\breve{\iota_{V_G}};(B_{G_1} \cap \breve{\Theta_1};n_x;\mathbb{T};n_y;\Theta_1);\iota_{V_G} \cup \breve{\kappa_{V_G}};B_{G_2};\kappa_{V_G}) && \text{Expand } \uplus \\
& \cup \breve{\iota_V};\Theta_1;\breve{\Theta_1};n_x;\mathbb{T};n_{e_2};\Theta_2;\kappa_{V_G} \cup \breve{\kappa_V};\Theta_2;\breve{\Theta_2};n_{a_2};\mathbb{T};n_y;\Theta_1;\iota_{V_G} \\
= \;& \Theta;\breve{\iota_{V_G}};(B_{G_1} \cap \breve{\Theta_1};n_x;\mathbb{T};n_y;\Theta_1);\iota_{V_G} \cup \Theta;\breve{\kappa_{V_G}};B_{G_2};\kappa_{V_G} && \text{Distribute } \Theta \\
& \cup \breve{\iota_V};\Theta_1;\breve{\Theta_1};n_x;\mathbb{T};n_{e_2};\Theta_2;\kappa_{V_G} \cup \breve{\kappa_V};\Theta_2;\breve{\Theta_2};n_{a_2};\mathbb{T};n_y;\Theta_1;\iota_{V_G} \\
= \;& (\breve{\iota_V};\Theta_1;\iota_{V_G} \cup \breve{\kappa_V};\Theta_2;\kappa_{V_G});\breve{\iota_{V_G}};(B_{G_1} \cap \breve{\Theta_1};n_x;\mathbb{T};n_y;\Theta_1);\iota_{V_G} \\
& \cup (\breve{\iota_V};\Theta_1;\iota_{V_G} \cup \breve{\kappa_V};\Theta_2;\kappa_{V_G});\breve{\kappa_{V_G}};B_{G_2};\kappa_{V_G} && \text{Expand } \Theta \\
& \cup \breve{\iota_V};\Theta_1;\breve{\Theta_1};n_x;\mathbb{T};n_{e_2};\Theta_2;\kappa_{V_G} \cup \breve{\kappa_V};\Theta_2;\breve{\Theta_2};n_{a_2};\mathbb{T};n_y;\Theta_1;\iota_{V_G} \\
= \;& \breve{\iota_V};\Theta_1;\iota_{V_G};\breve{\iota_{V_G}};(B_{G_1} \cap \breve{\Theta_1};n_x;\mathbb{T};n_y;\Theta_1);\iota_{V_G} \\
& \cup \breve{\kappa_V};\Theta_2;\kappa_{V_G};\breve{\iota_{V_G}};(B_{G_1} \cap \breve{\Theta_1};n_x;\mathbb{T};n_y;\Theta_1);\iota_{V_G} \\
& \cup \breve{\iota_V};\Theta_1;\iota_{V_G};\breve{\kappa_{V_G}};B_{G_2};\kappa_{V_G} \\
& \cup \breve{\kappa_V};\Theta_2;\kappa_{V_G};\breve{\kappa_{V_G}};B_{G_2};\kappa_{V_G} && \text{Distribution} \\
& \cup \breve{\iota_V};\Theta_1;\breve{\Theta_1};n_x;\mathbb{T};n_{e_2};\Theta_2;\kappa_{V_G} \cup \breve{\kappa_V};\Theta_2;\breve{\Theta_2};n_{a_2};\mathbb{T};n_y;\Theta_1;\iota_{V_G} \\
= \;& \breve{\iota_V};\Theta_1;\mathbb{I}_{V_1};(B_{G_1} \cap \breve{\Theta_1};n_x;\mathbb{T};n_y;\Theta_1);\iota_{V_G} \cup \breve{\kappa_V};\Theta_2;\mathbb{I}_{V_2};B_{G_2};\kappa_{V_G} && \text{Simplification} \\
& \cup \breve{\iota_V};\Theta_1;\breve{\Theta_1};n_x;\mathbb{T};n_{e_2};\Theta_2;\kappa_{V_G} \cup \breve{\kappa_V};\Theta_2;\breve{\Theta_2};n_{a_2};\mathbb{T};n_y;\Theta_1;\iota_{V_G} \\
= \;& \breve{\iota_V};\Theta_1;(B_{G_1} \cap \breve{\Theta_1};n_x;\mathbb{T};n_y;\Theta_1);\iota_{V_G} \cup \breve{\kappa_V};\Theta_2;B_{G_2};\kappa_{V_G} && \text{Identity of } ; \\
& \cup \breve{\iota_V};\Theta_1;\breve{\Theta_1};n_x;\mathbb{T};n_{e_2};\Theta_2;\kappa_{V_G} \cup \breve{\kappa_V};\Theta_2;\breve{\Theta_2};n_{a_2};\mathbb{T};n_y;\Theta_1;\iota_{V_G}
\end{aligned}
$$

Proof 3

$$B_2;\Theta_2 \subseteq \Theta_2;B_{G_2} \Rightarrow \kappa_V^{\smile};B_2;\Theta_2;\kappa_{V_G} \subseteq \kappa_V^{\smile};\Theta_2;B_{G_2};\kappa_{V_G}$$

Proof 4

$$\begin{aligned} & \mu;e_2^{\smile} = n_x;\mu;e_2^{\smile};n_{e_2} \subseteq n_x;\mathbb{T};n_{e_2} = \mathbb{I}_{V_1};n_x;\mathbb{T};n_{e_2} \subseteq \Theta_1;\Theta_1^{\smile};n_x;\mathbb{T};n_{e_2} \\ \Rightarrow\ & \iota_V^{\smile};\mu;e_2^{\smile};\Theta_2;\kappa_{V_G} \subseteq \iota_V^{\smile};\Theta_1;\Theta_1^{\smile};n_x;\mathbb{T};n_{e_2};\Theta_2;\kappa_{V_G} \end{aligned}$$

Proof 5

$$\begin{aligned} & a_2;\nu^{\smile} = n_{a_2};a_2;\nu^{\smile};n_y \subseteq n_{a_2};\mathbb{T};n_y = \mathbb{I}_{V_2};n_{a_2};\mathbb{T};n_y \subseteq \Theta_2;\Theta_2^{\smile};n_{a_2};\mathbb{T};n_y \\ \Rightarrow\ & \kappa_V^{\smile};a_2;\nu^{\smile};\Theta_1;\iota_{V_G} \subseteq \kappa_V^{\smile};\Theta_2;\Theta_2^{\smile}n_{a_2};\mathbb{T};n_y;\Theta_1;\iota_{V_G} \end{aligned}$$

Proof 6

$$\begin{aligned} & n_y;\mathbb{T};n_x \\ = \ & n_y;\mathbb{T};\mathbb{T};\mathbb{T};n_x \\ = \ & n_y;(\mathbb{T};n_y;\mathbb{T});\mathbb{T};(\mathbb{T};n_x;\mathbb{T});n_x && \text{Tarski rule} \\ = \ & (n_y;\mathbb{T};n_y);\mathbb{T};\mathbb{T};\mathbb{T};(n_x;\mathbb{T};n_x) \\ = \ & (n_y;\mathbb{T};n_y);\mathbb{T};(n_x;\mathbb{T};n_x) \\ = \ & (n_y;\Theta_1;\Theta_1^{\smile};n_y);\mathbb{T};(n_x;\Theta_1;\Theta_1^{\smile};n_x) \\ = \ & \Theta_1;\Theta_1^{\smile};n_y;\mathbb{T};n_x;\Theta_1;\Theta_1^{\smile} && \text{Proof } 7_{57} \end{aligned}$$

Then,

$$\begin{aligned} & (\Theta_1;\Theta_1^{\smile};n_y;\mathbb{T};n_x;\Theta_1);\Theta_1^{\smile} \subseteq n_y;\mathbb{T};n_x \\ \Leftrightarrow\ & \overline{n_x;\mathbb{T};n_y};(\Theta_1;\Theta_1^{\smile};n_y;\mathbb{T};n_x;\Theta_1) \subseteq \overline{\Theta_1} && \text{Theorem A.2}_{51} \\ \Leftrightarrow\ & \Theta_1^{\smile};\overline{n_x;\mathbb{T};n_y};\Theta_1 \subseteq \overline{\Theta_1^{\smile};n_x;\mathbb{T};n_y;\Theta_1} && \text{Theorem A.2}_{51} \\ \Rightarrow\ & \Theta_1^{\smile};B_1;\Theta \cap \Theta_1^{\smile};\overline{n_x;\mathbb{T};n_y};\Theta_1 \subseteq B_{G_1} \cap \overline{\Theta_1^{\smile};n_x;\mathbb{T};n_y;\Theta_1} && \text{Mono. of } \cap \\ \Rightarrow\ & \Theta_1^{\smile};(B_1 \cap \overline{n_x;\mathbb{T};n_y});\Theta_1 \subseteq B_{G_1} \cap \overline{\Theta_1^{\smile};n_x;\mathbb{T};n_y;\Theta_1} \\ \Rightarrow\ & \Theta_1;\Theta_1^{\smile};(B_1 \cap \overline{n_x;\mathbb{T};n_y});\Theta_1 \subseteq \Theta_1;(B_{G_1} \cap \overline{\Theta_1^{\smile};n_x;\mathbb{T};n_y;\Theta_1}) \\ \Rightarrow\ & (B_1 \cap \overline{n_x;\mathbb{T};n_y});\Theta_1 \subseteq \Theta_1;(B_{G_1} \cap \overline{\Theta_1^{\smile};n_x;\mathbb{T};n_y;\Theta_1}) && \mathbb{I} \subseteq \Theta;\Theta^{\smile} \end{aligned}$$

Recall monotonicity of ;, finally the following formula is proven,

$$\iota_V^{\smile};(B_1 \cap \overline{n_x;\mathbb{T};n_y});\Theta_1;\iota_{V_G} \subseteq \iota_V^{\smile};\Theta_1;(B_{G_1} \cap \overline{\Theta_1^{\smile};n_x;\mathbb{T};n_y;\Theta_1});\iota_{V_G}$$

Proof 7

Expanding $\mathsf{ran}\,(x;\Theta;\Theta^{\smile}) = x$ to

$$\Theta;\Theta^{\smile};x^{\smile};x;\Theta;\Theta^{\smile} \cap \mathbb{I} = x \tag{B.1}$$

Since x is a partial identity then Equ. $\mathrm{B.1}_{57}$ can be simplified to

$$\Theta;\Theta^{\smile};x;\Theta;\Theta^{\smile} \cap \mathbb{I} = x \tag{B.2}$$

Also $\Theta;\Theta^{\smile};x;\Theta;\Theta^{\smile}$ is an equivalent class since

$$\Theta;\Theta^{\smile};x;\Theta;\Theta^{\smile} = (\Theta;\Theta^{\smile};x^{\smile});(\Theta;\Theta^{\smile};x^{\smile})^{\smile} \tag{B.3}$$

$\Theta;\Theta^{\smile};x;\Theta;\Theta^{\smile}$ is less than or equal to $\Theta;\Theta^{\smile}$

$$\begin{aligned} & \Theta;\Theta^{\smile};x;\Theta;\Theta^{\smile} \\ \subseteq\ & \Theta;\Theta^{\smile};\mathbb{I};\Theta;\Theta^{\smile} \\ =\ & \Theta;\Theta^{\smile};\Theta;\Theta^{\smile} \\ =\ & \Theta;\Theta^{\smile} \end{aligned}$$

Based on Equ. B.2_{57}, it can be shown that $x;\Theta;\Theta^{\smile} = \Theta;\Theta^{\smile};x;\Theta;\Theta^{\smile}$,

$$\begin{aligned} & \Theta;\Theta^{\smile};x;\Theta;\Theta^{\smile} \cap \mathbb{I} = x \\ \Rightarrow\ & (\Theta;\Theta^{\smile};x;\Theta;\Theta^{\smile} \cap \mathbb{I});(\Theta;\Theta^{\smile}) = x;\Theta;\Theta^{\smile} \\ \Rightarrow\ & \Theta;\Theta^{\smile};x;\Theta;\Theta^{\smile};\Theta;\Theta^{\smile} \cap \mathbb{I};\Theta;\Theta^{\smile} = x;\Theta;\Theta^{\smile} \quad \text{[Kah01] Lemma A.2.1(v)} \\ \Rightarrow\ & \Theta;\Theta^{\smile};x;\Theta;\Theta^{\smile} \cap \Theta;\Theta^{\smile} = x;\Theta;\Theta^{\smile} \\ \Rightarrow\ & \Theta;\Theta^{\smile};x;\Theta;\Theta^{\smile} = x;\Theta;\Theta^{\smile} \end{aligned}$$

$\Theta;\Theta^{\smile};x;\Theta;\Theta^{\smile} = \Theta;\Theta^{\smile};x$ can be proven in the similar way. Then $x;\Theta;\Theta^{\smile} = x;\Theta;\Theta^{\smile};x = \Theta;\Theta^{\smile};x$ can be proven as well, due to the symmetric of proofs of two equalities, we only show a proof of one equality.

$$\begin{aligned} & x;\Theta;\Theta^{\smile};x \\ =\ & (x;\Theta;\Theta^{\smile});(\Theta;\Theta^{\smile}x) \\ =\ & (\Theta;\Theta^{\smile};x;\Theta;\Theta^{\smile});(\Theta;\Theta^{\smile};x;\Theta;\Theta^{\smile}) \\ =\ & \Theta;\Theta^{\smile};x;\Theta;\Theta^{\smile} \\ =\ & \Theta;\Theta^{\smile};x \end{aligned}$$

Appendix C

Haskell Implementation

A Haskell module is included; it implements the semantics of loop specification defined in Chapter 3_7. Also it implements the definition of extensible loop specification, composition and parameterised decomposition of extensible loop specification are implemented as well. The Fibonacci examples are created based on this module. Some of implementations are almost directly translated from the formal designs. The modules imported by this module are either from GHC standard library or from COCONUT() project.

```
module LoopSpec2 where
import CodeGraph
import CodeGraphOps
import Avoid
import Default
import PrelExts
import qualified LoopSpec as WT
import qualified Data.Map as Map
import qualified Data.Set as Set
import Data.List (partition, intersect, (\\), sort)
import Data.Maybe
```

The bifunctor cgTyParComp mentioned in Chapter 2_3 includes two components; the object component for the code graph category is just list concatenation.

```
cgTyParComp = (++)
```

The morphism component, cgParComp($\otimes$), is imported from module CodeGraphOps, which also provides morphism composition,cgComp(;), and

functions to create morphisms, like cgIdentity($\mathbb{I}$), cgSwap($\mathbb{X}$), cgTerm(!), cgCoTerm(i), cgDup(∇); the specification of these operators can be found in Chapter 2_3.

lhs2TeX is used to generate the pretty printed source code. In the previous paragraph, the operators are printed as the symbols in the parenthesis in order to make the implementation more close to the specification.

C.1 Loop Specification Definition

The following implements the loop specification definition Section 3.1_7. The Haskell presentation includes three parts, a code graph, an integer and a list of integers. The integer indicates where F starts in the input interface of the code graph, the list of integers are loop distances of F.

```
data LoopSpec n ty op = LoopSpec (CodeGraph n ty op) Int [Int]
```

C.2 Loop Specification Interface

The following provides a set of access functions to loop specification.

```
getCg :: LoopSpec n ty op → CodeGraph n ty op
getCg (LoopSpec cg _ _) = cg
getF, getC, getK :: (Ord n, Avoid n, Default n) ⇒ LoopSpec n ty op → [ty]
getF (LoopSpec cg n _)     = drop (n − 1) $ cgSrc cg
getK (LoopSpec cg n _)     = take (n − 1) $ cgSrc cg
getC ls@(LoopSpec cg n _) = take m $ cgTrg cg
  where m == length (cgTrg cg) − length (getF ls)
getD :: LoopSpec n ty op → [Int]
getD (LoopSpec _ _ d) = d
getN :: LoopSpec n ty op → Int
getN (LoopSpec _ n _) = n
```

C.3 Implementation of Loop Specification Semantics

Function mkSG and mkSGInv implement Def. 3.1_9. mkSG″ sorts F into two parts, trivial and non-trivial loop carried input, it returns the sorted interface

with the original interface, such that $(F, (A, B))$. mkSG′ generates a wiring code graph according to mkSG″, the first argument f decides the interface of the wiring code graph, besides the wiring code graph it also returns new list of loop distance according to shuffled interface. mkSG′ should be kept internal since an inappreciate f can break the function.

```
mkSG    :: (Ord n, Avoid n, Default n) ⇒
           LoopSpec n ty op → (CodeGraph n ty op, [Int])
mkSGInv :: (Ord n, Avoid n, Default n) ⇒
           LoopSpec n ty op → CodeGraph n ty op
mkSG    =      mkSG' id
mkSGInv = fst ∘ mkSG' swap
mkSG' :: (Ord n, Avoid n, Default n) ⇒
         (([n], [n]) → ([n], [n])) → LoopSpec n ty op →
         (CodeGraph n ty op, [Int])
mkSG' f ls@(LoopSpec cg n d) = (sg, shuffled_d)
  where
    shuffled_input = pupd2 (uncurry (++)) $ mkSG'' ls
    sg = uncurry cgShuffle (f shuffled_input) $ cgNodeType cg
    d_map = Map.fromList $ zip (fst shuffled_input) d
    shuffled_d = map (fromMaybe (error "mkSG'") ∘ flip Map.lookup d_map) $
                 snd shuffled_input
mkSG'' :: (Ord n, Avoid n, Default n) ⇒ LoopSpec n ty op → ([n], ([n], [n]))
mkSG'' (LoopSpec cg n d) = (loopCarriedInput, splitedLoopCarriedInput)
  where
    loopCarriedInput = drop (n − 1) $ cgInput cg
    splitedLoopCarriedInput = pupdD (map fst) $ partition ((≡ 1) ∘ snd) $
                              zip loopCarriedInput d
```

Function semtLoopSpec and gPrim are directly translated from the definition of $[\![\mathbf{G}]\!]_j$ and G', respectively, in Def. 3.2_{10}

```
semtLoopSpec :: (Ord n, Avoid n, Default n, Eq op, Show ty, Eq ty) ⇒
                LoopSpec n ty op → Int → CodeGraph n ty op
semtLoopSpec ls n = (𝕀k ⊗ sg) ⨟ select ⨟ (𝕀k ⊗ sgInv) ⨟ cg
  where
    cg    = getCg ls
    k     = getK ls
    b     = cgNodeTypes cg bNode
    a     = cgNodeTypes cg aNode
```

```
    g'      = semtLoopSpec (gPrim ls) (pred n)
    sgInv   = mkSGInv ls
    (sg, _) = mkSG ls
    (aNode, bNode) = snd $ mkSG'' ls
    firstIte  == I_{k⊗a} ⊗ !_b ⊗ i_b
    otherIte == I_{k⊗a} ⊗ i_b ⊗ I_b ; g' ; I_{k⊗a} ⊗ I_b ⊗ !_b
    select    == if n ≡ 1 then firstIte else otherIte
gPrim :: (Ord n, Avoid n, Default n, Eq op, Show ty, Eq ty) ⇒
      LoopSpec n ty op → LoopSpec n ty op
gPrim ls =  LoopSpec cg' n d'
  where
    k       == getK ls
    c       == getC ls
    b       == cgNodeTypes cg bNode
    a       == cgNodeTypes cg aNode
    n       == getN ls
    cg      == getCg ls
    sgInv   == mkSGInv ls
    aTIMEb == cgTrg sg
    (aNode, bNode) = snd $ mkSG'' ls
    (sg, d'_of_AxB) = mkSG ls
    cg' = (∇_k ⊗ I_{a⊗b} ; I_k ⊗ (I_k ⊗ sgInv ; cg ; !_c ⊗ sg)) ⊗ I_b ; I_{k⊗a} ⊗ X_{b,b}
    d'_of_B' = dropWhile (≡ 1) d'_of_AxB
    d'      = (map (const 1) d'_of_AxB) ++ (map pred d'_of_B')
```

C.4 Extensible Loop Specification Definition

The Haskell representation of an extensible loop specification includes four parts, a code graph, number of constant inputs nk, number of control outputs nc and a list of loop distance d. E_{in} and E_{out} can be calculated based on these information.

```
data ELoopSpec n ty op = ELoopSpec{
  cg :: (CodeGraph n ty op),
  nk :: Int,
  nc :: Int,
  d  :: [Int]}
```

C.5 Convert from Loop Specification

Function fromWTls converts the loop specification definition created by Wolfgang Thaller to extensible loop specification.

```
fromWTls :: WT.LoopSpec n ty op → ELoopSpec n ty op
fromWTls wtls = let wtlsFin  = WT.lsFinputs wtls
                    wtlsFout = WT.lsFoutputs wtls
                    wtlsK    = WT.lsKinputs wtls
                    wtlsC    = WT.lsCoutputs wtls
  in ELoopSpec{ nk = length wtlsK
              , nc = length wtlsC
              , d  = map negate $ WT.lsDs wtls
              , cg = cgUpdateOutput (const (wtlsC ++ wtlsFout)) $
                     cgUpdateInput (const (wtlsK ++ wtlsFin)) $
                     WT.lsCg wtls
              }
```

C.6 Extensible Loop Specification Access Functions

Note: getEFTy(F) calculates loop carried interface, F, by just using input interface of the code graph, it also can use output interface of the code graph to calculate F.

```
getEin, getEout, getEK, getEC, getEFin, getEFout
   :: (Ord n, Avoid n, Default n) ⇒ ELoopSpec n ty op → [x] → [x]
getEin    (ELoopSpec cg n  _  d) = drop (n + length d)
getEout   (ELoopSpec cg _  n  d) = drop (n + length d)
getEK     (ELoopSpec cg nk _  _) = take nk
getEC     (ELoopSpec cg _  nc _) = take nc
getEFin   (ELoopSpec cg nk _  d) = take (length d) ∘ drop nk
getEFout  (ELoopSpec cg _ nc  d) = take (length d) ∘ drop nc
```

E_{in} , E_{out} , K , C , F

```
   :: (Ord n, Avoid n, Default n) ⇒ ELoopSpec n ty op → [ty]
```

K_{els} = getEK els \$ cgSrc \$ G_{els}

C_{els} = getEC els \$ cgTrg \$ G_{els}

F_{els} = getEFin els \$ cgSrc \$ G_{els}

```
E_in els  = getEin  els $ cgSrc $ G_els
E_out els = getEout els $ cgTrg $ G_els
getEinNode, getEoutNode, getEFinNode, getEFoutNode, getECNode
   :: (Ord n, Avoid n, Default n) ⇒ ELoopSpec n ty op → [n]
getEinNode    els = getEin    els $ cgInput $ G_els
getEoutNode   els = getEout   els $ cgOutput $ G_els
getEFinNode   els = getEFin   els $ cgInput $ G_els
getEFoutNode  els = getEFout  els $ cgOutput $ G_els
getECNode     els = getEC     els $ cgOutput $ G_els
getEKNode     els = getEK     els $ cgInput $ G_els
getNK = nk
getNC = nc
getED = d
G     = cg
getFoutPos :: (Ord n, Avoid n, Default n) ⇒
              ELoopSpec n ty op → Int → Int
getFoutPos els fin = fin − getNK els + getNC els
getFoutPosList :: (Ord n, Avoid n, Default n) ⇒
  ELoopSpec n ty op → [Int] → [Int]
getFoutPosList els = map (getFoutPos els)
```

C.7 Composition of Extensible Loop Specification

⊛ is translated directly from Def. 4.2_{21}.

```
⊛ ::(Ord n, Avoid n, Default n, Show ty, Eq ty, Eq op) ⇒
    ELoopSpec n ty op → ELoopSpec n ty op → ELoopSpec n ty op
a ⊛ b = ELoopSpec cg nk nc d
  where
    ka = K_a
    kb = K_b
    fa = F_a
    fb = F_b
    kb_fb = kb ⊗ fb
    ca = C_a
    cb = C_b
```

$$
\begin{aligned}
\mathsf{nk} &= \mathsf{getNK\ a} + \mathsf{getNK\ b} \\
\mathsf{nc} &= \mathsf{getNC\ a} + \mathsf{getNC\ b} \\
\mathsf{d} &= \mathsf{getED\ b} +\!\!+ \mathsf{getED\ a} \\
\mathsf{cg} &= \mathsf{a1} \fatsemi \mathsf{a2} \fatsemi \mathsf{a3} \fatsemi \mathsf{a4} \\
\mathsf{a1} &= \mathbb{I}_{\mathsf{ka}} \otimes \mathbb{X}_{\mathsf{kb\_fb},F_{\mathsf{a}} \otimes E_{in\,\mathsf{a}}} \\
\mathsf{a2} &= G_{\mathsf{a}} \otimes \mathbb{I}_{\mathsf{kb\_fb}} \\
\mathsf{a3} &= \mathbb{I}_{\mathsf{ca} \otimes \mathsf{fa}} \otimes (\mathbb{X}_{E_{in\,\mathsf{b}},\mathsf{kb\_fb}} \fatsemi G_{\mathsf{b}}) \\
\mathsf{a4} &= \mathbb{I}_{\mathsf{ca}} \otimes \mathbb{X}_{\mathsf{fa,cb} \otimes \mathsf{fb}} \otimes \mathbb{I}_{E_{out\,\mathsf{b}}}
\end{aligned}
$$

C.8 Implementing the First Step of Decomposition

This section implements the algorithm specified in Subsection 4.4.3$_{29}$. The following two auxiliary functions, trg′ and src′, directly implement the specification in Subsection 4.4.3$_{29}$.

```
trg' :: (Ord op, Ord n, Avoid n, Default n, Show ty, Eq ty, Eq op, Show n) ⇒
        CodeGraph n ty op → Set.Set Edge → Set.Set n
trg' cg = Set.fold (Set.union ∘ Set.fromList ∘ cgEdgeResults cg) Set.empty
src' :: (Ord op, Ord n, Avoid n, Default n, Show ty, Eq ty, Eq op, Show n) ⇒
        CodeGraph n ty op → Set.Set Edge → Set.Set n
src' cg = Set.fold (Set.union ∘ Set.fromList ∘ cgEdgeArgs cg) Set.empty
```

Function eLoopSpecDecomp implements the first step of decomposition algorithm. It takes two inputs, an extensible loop specification and an interface of right extensible loop specification; the interface is specified by two lists of integers, the integers are positions in input or output interface of the original extensible loop specification, the first list specifies the input interface and the second list only specifies the control outputs.

```
eLoopSpecDecomp :: (Ord op, Ord n, Avoid n
                   , Default n, Show ty
                   , Eq ty, Eq op, Show n) ⇒
                   ELoopSpec n ty op → ([Int], [Int]) →
                   (ELoopSpec n ty op,  (ELoopSpec n ty op
                                        , ELoopSpec n ty op))
eLoopSpecDecomp els (right, rightC) = (midELoopSpec,
                                       (leftELoopSpec, rightELoopSpec))
```

```
where
  errStr   = "eLoopSpecDecomp:"
  inputs   = cgInput $ G_els
  outputs  = cgOutput $ G_els
  elsFin   = getEFinNode els
  elsFout  = getEFoutNode els
  elsEin   = getEinNode  els
  elsEout  = getEoutNode els
  elsK     = getEKNode  els
  elsC     = getECNode  els
  elsCG    = G_els
  elsEdges = cgEdgeList  elsCG
  elsNodes = cgNodeList elsCG
  ioMap  = mkELoopSpecIOMap els
  dMap   = mkELoopSpecLDMap els
  diff   = listAsSet Set.difference
  union  = listAsSet Set.union
  rmDup = Set.toList ∘ Set.fromList
  rightFoutPos = getFoutPosList els right


  rightNodesIn    = map (inputs!!) right
  rightNodesCout' = map (outputs!!) rightC
  rightNodesFin   = rightNodesIn `intersect` elsFin
  rightNodesFout  =
    map (lookupWithErrorStr (errStr ++ show ioMap) ioMap)
        rightNodesFin
  rightNodesK     = rightNodesIn `intersect` elsK
  rightNodesC     = rightNodesCout' `intersect` elsC
  rightNodesEout  = elsEout
  rightNodesEin   = sort $ (concatMap (cgEdgeArgs elsCG)
                                      rightDescEdges ++
                                      rightNodesFout ++
                                      rightNodesC ++
                                      rightNodesEout)
                           `diff` rightDescNodes
  (rightDescNodes, rightDescEdges) =
    cgBFS elsCG (rightNodesFin ++ rightNodesK)
  rightNodesOut = rightNodesC  ++  rightNodesFout ++ rightNodesEout
```

```
rightNodes      = rightNodesFin ++  rightNodesK   ++ rightNodesEin ++
                  rightNodesOut ++  rightDescNodes
invRightDescEdges = elsEdges \\ rightDescEdges
invRightNodes     = elsNodes \\ rightNodes
rightCG''  = cgDelNodes (cgDelEdges invRightDescEdges elsCG) invRightNodes
rightCG'   = cgUpdateInput (const  (rightNodesK ++
                                    rightNodesFin ++
                                    rightNodesEin))
                           rightCG''
rightCG    = cgUpdateOutput (const rightNodesOut) rightCG'
rightELoopSpec =
  els  {cg = rightCG
       ,nk = length rightNodesK
       ,nc = length rightNodesC
       ,d  = map (lookupWithErrorStr errStr dMap) rightNodesFin
       }
```

Since there is possible to have duplication in output interface node list, leftNodesOut, leftNodesFout, leftNodesC are calculated in a complicated way.
For the input interface node list, since there is no duplication, then \\ , i.e., list difference, is safe to use.

```
leftNodesIn     = inputs \\ rightNodesIn
leftNodesOut    = map (outputs!!) (([0 .. length outputs - 1] \\
                                    rightFoutPos) \\
                                    rightC)
leftNodesFin    = elsFin \\ rightNodesFin
leftNodesFout   = map (outputs!!) ([getNC els .. length outputs - 1] \\
                                    rightFoutPos)
leftNodesK      = elsK \\ rightNodesK
leftNodesC      = map (outputs!!) ([0 .. getNC els] \\ rightC)
leftNodesEin    = elsEin
leftNodesEout1  = (concatMap (cgEdgeResults elsCG)
                             leftAsceEdges ++
                             leftNodesFin ++
                             leftNodesK ++
                             leftNodesEin)
                  `diff` leftAsceNodes
leftNodesEout = sort $  leftNodesEout1 `union`
                        (filter (flip elem leftNodes) $
```

```
                                    concatMap (cgEdgeArgs elsCG) invLeftAsceEdges ++
                                    rightNodesOut
                                  )
    invLeftAsceEdges = elsEdges \\ leftAsceEdges
    invLeftNodes     = elsNodes \\ leftNodes
    (leftAsceNodes, leftAsceEdges) =
      cgRBFS elsCG (leftNodesFout ++ leftNodesC)
    leftNodes = leftNodesFin ++ leftNodesFout ++ leftNodesK ++ leftNodesEout1 ++
                leftNodesEin ++ leftAsceNodes ++ leftNodesC
    leftCG'' = cgDelNodes (cgDelEdges invLeftAsceEdges elsCG) invLeftNodes
    leftCG'  = cgUpdateInput  (const (leftNodesK ++
                                      leftNodesFin ++
                                      leftNodesEin)) leftCG''
    leftCG   = cgUpdateOutput (const (leftNodesC ++
                                      leftNodesFout ++
                                      leftNodesEout)) leftCG'

    leftELoopSpec =
      els {cg = leftCG
          ,nk = length leftNodesK
          ,nc = length leftNodesC
          ,d  = map (lookupWithErrorStr errStr dMap) leftNodesFin
          }

    invMidNodes = rmDup $ (leftNodes ++ rightNodes) `diff`
                          (leftNodesEout ++ rightNodesEin)
    midCG'' = cgDelEdges (leftAsceEdges ++ rightDescEdges) elsCG
    midCG'  = cgDelNodes midCG'' invMidNodes
    midCG   = cgUpdateInput (const leftNodesEout) $
              cgUpdateOutput (const rightNodesEin) midCG'
    midELoopSpec =
      els {cg = midCG
          ,d  = []
          ,nk = 0
          ,nc = 0
          }
```

mkELoopSpecIOMap creates loop carried input-output relation.

```
mkELoopSpecIOMap :: (Ord n, Default n, Avoid n) ⇒
                    ELoopSpec n ty op → Map.Map n n
```

```
mkELoopSpecIOMap els =
  Map.fromList $ zip (getEFinNode els) (getEFoutNode els)
```

mkELoopSpecLDMap creates a map, maps F to loop distance

```
mkELoopSpecLDMap :: (Ord n, Default n, Avoid n) ⇒
                    ELoopSpec n ty op → Map.Map n Int
mkELoopSpecLDMap els = Map.fromList $ zip (getEFinNode els) (d els)
```

C.9 Implementing the Second Step of Decomposition

Function eLoopSpecDecompM implements the second step of the decomposition algorithm presented in Subsection $4.4.4_{31}$. It takes two arguments, a extensible loop specification and a set of edges, the set of edges contains all edges which is all edges of M_L, then all edges of M_R can be calculated.

```
eLoopSpecDecompM :: (Ord op, Ord n, Avoid n
                    , Default n, Show ty
                    , Eq ty, Eq op, Show n) ⇒
                    ELoopSpec n ty op → Set.Set Edge →
                    Maybe (ELoopSpec n ty op, ELoopSpec n ty op)
eLoopSpecDecompM m eGML =
if ¬ $ Set.null $ (trg' cg eGMR ∩ src' cg eGML)
then Nothing
else Just (mL, mR)
  where
    cg = Gm
    e = cgEdgeSet cg
    eGMR = e − eGML
    nGML = Set.fromList (cgInput cg) ∪ src' cg eGML ∪ trg' cg eGML
    hN   = sort $ Set.toList $
             nGML ∩ (src' cg eGML ∪ Set.fromList (cgOutput cg))
    gML =  cgDelNodes (cgDelEdgeSet eGMR cg) $ Set.toList $
           (cgNodeSet cg) − nGML
    mL   = ELoopSpec{ nk = 0
                    , nc = 0
                    , d  = []
```

```
                    , cg = cgUpdateOutput (const hN) $ gML
                    }
gMR = cgDelNodes (cgDelEdgeSet eGML cg) $ Set.toList $
      nGML − (Set.fromList hN)
mR =  ELoopSpec{ nk = 0
                    , nc = 0
                    , d  = []
                    , cg = cgUpdateInput (const hN) $ gMR
                    }
```

Bibliography

[AK08a] Christopher K. Anand and Wolfram Kahl. Coconut – code-graph-centered parallelisation. Presentation at CASCON 2008 Workshop on Compiler-Driven Performance, October 2008. `http://www.eecg.toronto.edu/~steffan/workshops/08/cdp/`.

[AK08b] Christopher Kumar Anand and Wolfram Kahl. Code graph transformations for verifiable generation of SIMD-parallel assembly code. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance, AGTIVE 2007*, volume 5088 of *LNCS*, 2008. (to appear).

[CBS96] Jordi Cortadella, Rosa M. Badia, and Fermín Sánchez. A mathematical formulation of the loop pipelining problem. In *XI Conference on Design of Integrated Circuits and Systems*, pages 355–360, Barcelona, November 1996.

[CG99] A. Corradini and F. Gadducci. An algebraic presentation of term graphs, via gs-monoidal categories. *Applied Categorical Structures*, 7(4):299–331, 1999.

[KAC06a] Wolfram Kahl, Christopher Kumar Anand, and Jacques Carette. Control-flow semantics for assembly-level data-flow graphs. In Wendy MacCaull, Michael Winter, and Ivo Düntsch, editors, *RelMiCS 2005*, volume 3929 of *LNCS*, pages 147–160. Springer, 2006.

[KAC06b] Wolfram Kahl, Christopher Kumar Anand, and Jacques Carette. Control-flow semantics for assembly-level data-flow graphs. In Wendy McCaull et al., editors, *8th Intl. Sem. Relational Methods in Computer Science*, volume 3929 of *LNCS*, pages 147–160. Springer, 2006.

[Kah01] Wolfram Kahl. A relation-algebraic approach to graph structure transformation, 2001. Habil. Thesis, Informatik, UniBw München, Techn. Ber. 2002-03.

[Kah06] Wolfram Kahl. Semigroupoid interfaces for programming with relations in Haskell. In Renate Schmidt and Georg Struth, editors, *Relations and Kleene Algebra in Computer Science, RelMiCS/AKA 2006*, volume 4136 of *LNCS*, pages 235–250. Springer, 2006. (to appear).

[Koz94] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Infor. and Comput.*, 110(2):366–390, May 1994.

[ORSA05] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, November 2005.

[RVVA04] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. Decoupled software pipelining with the synchronization array. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 177–188, Washington, DC, USA, 2004. IEEE Computer Society.

[SHW97] Gunther Schmidt, Claudia Hattensperger, and Michael Winter. Heterogeneous relation algebra. In Chris Brink, Wolfram Kahl, and Gunther Schmidt, editors, *Relational Methods in Computer Science*, Advances in Computing, chapter 3, pages 39–53. Springer-Verlag, Wien, New York, 1997. ISBN 3-211-82971-7.

[SS93] Gunther Schmidt and Thomas Ströhlein. *Relations and Graphs, Discrete Mathematics for Computer Scientists.* EATCS-Monographs on Theoretical Computer Science. Springer, 1993.

[Tha06] Wolfgang Thaller. Explicitly staged software pipelining. Master's thesis, McMaster University, Department of Computing and Software, 2006. http://sqrl.mcmaster.ca/~anand/papers/ThallerMScExSSP.pdf.

Index