Formalization of a Reverse Engineering Strategy

Gustavo Villavicencio

October 27, 2002

Submitted to the
Department of Computer Sciences
School of Physics, Mathematics and Natural Sciences
of the National University of San Luis
in fulfillment with the requirements
for the degree of Master in Software Engineering

Copyright © 2002 Gustavo Villavicencio

Thanks, Mom!

Contents

1 Introduction **1**

1.1 Structure of thesis - Thesis Outline 3

2 Background **5**

2.1 Reverse Engineering 5

2.2 Program slicing 6

2.3 Theoretic Frame 8

2.3.1 Formal methods 8

2.3.2 Functional programming 10

2.3.3 Denotational / Operational Semantics 13

2.4 Summary 15

3 Algebra of Programming **17**

3.1 Basic concepts 17

3.2 Products toolbox 19

3.3 Co-products toolbox 21

3.4 Conditionals 23

3.5 Recursion 26

3.6 Anas, catas and hylomorphisms: An overview 29

3.7 Monads 33

4 Formal Reverse Engineering **35**

4.1 Approaches to Formal Reverse Engineering 35

4.1.1 Approach I 35

4.1.2 Approach II 37

4.2 Analysis 44

4.3 Summary 45

5 Formalization of a Strategy for Reverse Engineering **47**

5.1 Introduction 47

5.2 Scope 47

5.3 Overview 47

5.4 The **RPC** process 48

5.4.1 Non terminate situations 50

5.4.2 Accumulation parameter introduction 52

5.5 Conditioned program slicing 52

5.6 The role of algebra 52

5.7 Case Studies 54

5.8 Case study I 54

5.9 Case Study II: Monads. 62

5.10 Case study III: Conditional slicing in the reverse calculation process 67

5.11 Evaluation 78

5.12 Summary 78

6 Conclusions and Future Work **79**

A Appendix A: Source Code Examples **81**

A.1 Example I 81

A.2 Example II: Monads 82

A.3 Example III: Pointers 85

B Appendix B: Calculated Slices **89**

B.1 Calculated slices from Example I 89

B.2 Dos to Unix: Monads 91

B.3 Removing string.(Pointers) 102

B.4 Conditioned slicing 107

Bibliography **113**

No hay orden establecido que sea duradero sino el que une el principio con el fin en un ciclo inmutable.

La consolación de la filosofía

Boecio (480-524 D.C.)

Chapter 1

Introduction

In chemistry a sample that requires analysis is often a mixture of many components in a complex matrix. For samples containing unknown compounds, the components must be separated from each other for each individual component to be identified by different analytical methods. Some separation methods are: *centrifugation*, *chromatography*, *crystalization*, *distillation*, etc. In general, the separation process is executed under certain conditions. Thus, for example, the *gas chromatographic* process to separate volatile organic compounds is performed as follows: Mobile phases are generally inert gases such as helium, argon, or nitrogen. The injection port consists of a rubber septum through which a syringe needle is inserted to inject the sample. The injection port is maintained at a higher temperature than the boiling point of the least volatile component in the sample mixture. Since the partitioning behavior is dependent on temperature, the separation column is usually contained in a thermostat-controlled oven. The separation of components with a wide range of boiling points is accomplished by starting at a low oven temperature and increasing the temperature over time to elude the high-boiling point components. Most columns contain a liquid stationary phase on a solid support. Separation of low-molecular weight gases is accomplished with solid adsorbents [45].

In physics, *mass spectrography* separates electrically charged particles according to their mass. A mass spectrograph is often used to measure the mass of isotopes as follows: The mass spectrograph separates molecules on the basis of mass to charge ratios. A gas sample is aspirated into a high vacuum chamber (10^{-5} mmHg) where an electron beam ionizes and fragments the components of the sample. The ions are accelerated by an electric field into a final chamber, which has a magnetic field, perpendicular to the path of

the ionized gas stream. In the magnetic field the particles follow a path wherein the radius of curvature is proportional to the charge: mass ratio. A detector plate allows for the determination of the gas components and for each component's concentration [24].

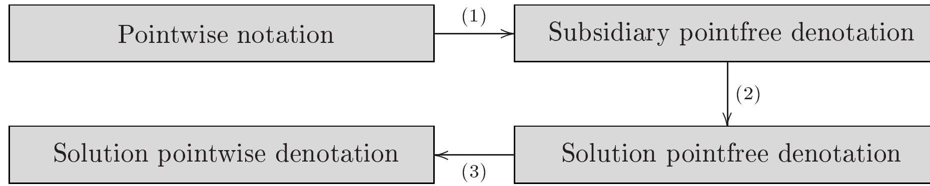
In the same way as in chemistry and physics, reverse engineering applies *slicing technique* to decompose the source code in smaller fragments (a more accurate parallelism is presented in [39]). This decomposition is usually carried out in order to understand unknown source code, and then perform *maintenance*, *re-engineering*, *migration*, etc. Each fragment is calculated under certain conditions. Such conditions are defined in the *slicing criterion*. Section (2.2) shows a spread presentation on this technique.

But the source code decomposition itself is not sufficient to acquire an adequate understanding of the source code. The obtainment of a more abstract representation is also necessary. Graphical views like data-flow or control-flow diagrams are often used. However, these representations do not improve the understanding of the process:

- The functional inferences carried out from them are intuitive, and thus new errors can be introduced.
- They are not appropriate support for the final specification, and usually, only an informal (textual) specification can be obtained through them.
- The specification integration (composition) is not carried out by reliable means.

To improve this situation, especially when reverse engineering is practiced on a critical system, a mathematically founded systematic method is required. Specifically, in this work we are proposing the application of the *algebra of programming* [3] as a calculus mechanism. In this strategy, the specifications are literally calculated from the source code, and so their accuracy, non ambiguity, and capacity of precise composition are guaranteed. After the slice extraction, their denotational semantics will be expressed in **HASKELL** or **VDM-SL** [26], in which some transformations will be executed.

But the expressions obtained in *pointwise notation* will not be abstract enough [39]. Therefore they will be translated into *pointfree style* [41] ((1) in the next picture) where the notation is more compact and so the "reasoning" is less complicated. The calculus process will continue at this pace until appropriate (expressions without variables, logical connectives, quantifiers, etc.) specifications are obtained (2). Finally, the last expression will be translated again into pointwise notation(3).



In brief, the strategy applied is the *Laplace transformation*.

Some intent to apply this approach has already been carried out on the data domain [37, 33]. In the present work we are trying to extend its application area to the more complex domain of algorithmical structures.

1.1 Structure of thesis - Thesis Outline

As we stated in the previous section, in this thesis we propose a new strategy for reverse engineering, more precisely, for formal reverse engineering. However, we prefer to refer to *reverse program calculation* (**RPC**). The innovation is introduced in the application of the slicing technique to reduce the complexity, and in the use of algebra of programming in the reverse calculus of specifications from the extracted slices.

Essentially, the structure of this work is very simple. In chapter 2 we present all the topics that integrate the conceptual base of this strategy. However, we have considered important to dedicate a special chapter to the algebra of programming because its laws are in the visible surface of the reverse calculus process. So chapter 3 is dedicated to it. In chapter 4, the two most accepted formal reverse engineering strategies are described and analyzed. Chapter 4 presents the proposed approach. Finally, in chapter 5 we present the conclusions and the future work.

The work is also complemented with appendixes with the complete source code for each case analyzed in chapter 4. The final appendix also shows the calculated slices.

Chapter 2

Background

2.1 Reverse Engineering

Terminology

Data versus programs cf DRE, Karma, abstraction/representation, formal methods etc

Cf. A Calculational Approach to Reverse Specification . Seminar at UNU/IIST, Macau, 13 of May 1997. (”Information system slicing”)

Reverse engineering is the process by which specifications, design and documentation are recovered from a source code. In [7] it is defined as the process to

- identify the system’s components and their interrelationships and
- create representation of the system in another form or at a higher level of abstraction

In a normal situation, source code modification as a consequence of maintenance activities is not reflected in the documentation. Thus, the real source from which to start the reverse engineering process (**REP**) is an existing functional system.

Obviously, the analysis of hundreds of thousands of lines written by unknown programmers is not easy. Automatic tools are necessary to perform this work; the reverse engineering (**RE**) area is known for the abundant development of tools.

According to the definition, **REP** does not involve modifications or creating a new system based on the reverse-engineered subject system. It is a process of examination only.

The information obtained is used for constructing a higher level of abstraction. Representation becomes important at this point. The format of the information acquired by automatic and semi-automatic means is relevant to facilitate the understanding of a legacy system. Graphical representation is the most used: control-flow diagrams, dataflow diagrams, structure charts, entity-relationship diagrams, etc. But non-graphical representation (textual), is used as well.

In the most accepted definition of **RE** given in [7], the specification recovery is also considered. Unlike other representations, the format of the specifications is non-graphical (textual), and it is acquired by semi-automatic means.

The output of **REP**, is applied in multiple activities: maintenance, reuse, construction of new systems, software renovation, etc.

On the other hand, it is necessary to point out that the techniques and methods used in **REP** can be formal, semi-formal and informal depending on the mathematical foundation. So the extraction of functionality reading the source code is informal, but by means of code slicing it is semi-formal, and by applying pre and postconditions it is formal. The last methods will be studied in more detail later on.

2.2 Program slicing

Program slicing is a program decomposition technique that extracts a set of sentences related to a specific computation. One of the most usual questions during maintenance is, what program statements affect the variable v at statement s? The program slicing technique brings the solution. The previous sentences at n which affect v are calculated applying dependency analysis . Thus, one gets an easier to understand reduced view of the program composed only of relevant statements. The pair $< n, v >$ is called *slicing criterion*.

The first idea of a program slice introduced in [56] was the *executable backward static slices*. Backward because in the intermediate representation (dependence graph) used, the edges are traversed in the reverse direction starting from n. Static because the calculus process is only based on static dependencies, and no data inputs are considered.

After the seminal work [56], abundant bibliography on the applications of this technique was produced: software maintenance [11], debugging, testing, program understanding [23, 28], reuse [8], parallelization, software metrics, formal specification recovery [40].

Different forms of slices can be calculated as well [22]. Opposed to a backward slice, a *forward slice* extracts those sentences affected by the value of v at sentence n [25]. The notion of *dynamic slice* was introduced in [27]. As a complement of slicing criterion, input information is provided to the program for a specific execution. This additional information allows the calculation of shorter slices than static slices. Therefore dynamic slices are appropriate for testing and debugging, and less useful for understanding and reuse [22]. The *amorphous slice* [21] is another form of slice. Unlike traditional slice (backward static slice), amorphous slice by means of simplifying transformation improves the simplification power of the traditional slice while preserving a projection of its semantics. Therefore the resulting slice is more useful for understanding purposes.

Conditioned slice [5] is another important variant of the original definition. Like static slices, conditioned slices preserve the behavior of the source program with respect to the slicing criterion for any program execution. But a first order logic formula is defined on the data input, so the set of initial states of the program that characterize the executions is specified. Formally:

$$C = (F(V_{in}), p, V) \tag{2.1}$$

where F is the first order logic formula on the variables in V_{in}, p is a statement in a program P and V is a subset of the variables in P. A way to compute conditional slices is to reduce the program by imposing an input condition in order to generate a *conditioned program*. The static slices are calculated on this conditioned program. The calculation of the conditioned program is usually made by means of a *symbolic executor*. In this context, symbolic expressions are used to assign values to the variables.

In this work we shall apply the backward static slicing technique and we shall present a case where we apply conditioned slices. In order to calculate conditioned slices, the strategy briefly described previously will be applied. The use of different kinds of slicing techniques demonstrates the strenght of the strategy that we are proposing: The flexibility to articulate different slicing techniques with the algebra of programming.

The application of other techniques, *e.g.* amorphous slices, would be an extension of the current work.

2.3 Theoretic Frame

2.3.1 Formal methods

Formal, semi-formal, combinations of

On the first days of software engineering, the development process was made ad hoc. But as soon as the complexity increased, the necessity to produce software in a systematic way emerged. However, the systematic way of constructing software is no guarantee of obtaining correct and reliable products. The application of methods, techiques and tools with mathematical foundation is also necessary . Again, it is not the final solution (as suggested in [20]), but these can greatly increase our understanding of a system by revealing inconsistencies, ambiguities and incompleteness that might otherwise pass undetected [9].

From the software engineering point of view, *formal methods* are mathematically based techniques to precisely describe a software system. In [4] a more precise definition is given: By formal methods we understand a set of Formal Specification and Design Calculi techniques (a particular case of these is the *reification calculi* [34, 36]). Where design calculi is a set of proof or specification transformation rules.

The use of formal languages to construct specifications make them more concise and less ambiguous, and it also allows the gradual introduction of more concrete details until an implementation is achieved.

These techniques are applied throughout the development of a system and involve the use of *refinement (or reification) techniques* [29, 30] (design calculi technique in the previous definition) and proofs of correctness at each stage to ensure that the current specification completely and correctly refines the previous specification. Refinement is an important concept in formal methods that involves the construction of specifications at different levels of abstraction, and also the demonstrations that ensure a specification at a lower level satisfies the requirements imposed by another at a higher level.

But this style "invent-and-verify" is impractical due to the mathematical complexity involved in real life software problems [36]. In the last years, the state of the art has evolved and a new style for software design has arisen: "correct by construction". The central matter in this strategy is to avoid the proof by replacing it with structural calculation. The reification process is executed based on available laws in calculus. One of the best known reification processes is supported by **Set** calculus [34, 36, 32]. Since the strategy of reverse program calculation is based on **Set** calculus, it will be detailed later on.

But, what about the automatic support of the reification process? Up to date, there is a plethora of notations and methods to be used, among the most popular and with industrial experience we have **Z**, **VDM**(Viena Development Method), **RAISE**(Rigorous Approach to Industrial Software Engineering), **LOTOS**, etc. The first two are model-based specification languages that focus on specifying the behavior of sequential systems. States are described in terms of mathematical structures like sets, relations and functions, and actions that can change or observe the component's internal state in terms of defined pre- and post-conditions [9, 32]. Both are based on set theory and first order predicate calculus, and both are appropriate to specify the functional aspects of a software system. On the other hand, **LOTOS** and **RAISE** are based on **CSP**(communicating sequential processes), which makes the synchronized communication between processes possible.

We can see that each method has specific characteristics that make it applicable to specific domain problems. However, no method can model all aspects of a system [1].

But, why recover formal specifications by a reverse program calculation (**RPC**) process? The fact of having formal specifications expressed in a formal language implies that we have an accurate, comprehensive and consistent representation of the software system. Since **RPC** process has a strong mathematical base, the specifications obtained have the properties indicated before. Obviously, these characteristics cannot be expected in representations obtained by informal **REP**(recall 2.1).

There will be systems to be reversed (maintained, renovated, etc.) that will not permit the recovery of erroneous representations. This is certain, especially on critical software systems. It is evident that the more critical the software system to be analyzed is, the more the need for applying a formal **REP**.

Besides, like formal forward engineering, formal reverse engineering must get closer to the programmers. In the last years the formal methods community has made a great effort in taking this technology from the university to the industry. The same must happen with formal reverse engineering in the near future.

VDM, Z, etc etc browse:

`http://www.di.uminho.pt/~jno/FME-SoE/`

`http://www.afm.sbu.ac.uk/`

Either here or in the conclusions: relate to the work of

`http://www.ist.tu-graz.ac.at/aichernig/` on Testing and Formal Methods.

2.3.2 Functional programming

Traditional programming languages (e.g. Pascal, C, COBOL, Fortran) rely heavily on modifying a *state* (collection of variables). If we disregard that the program can be executed through a sequence of changes of states, we shall see that there is a more abstract way of interpreting a program.

The states are usually modified by assignment commands which can be performed in sequential, conditional or iterative form, depending on some properties of the current state.

Functional programming adopts a different perspective to that of this model. A functional program is simply an expression, and execution means evaluation of the expression. As Jeremy Gibbons points out in [18]

> Expressions in a functional programming language behave as they do in ordinary mathematics, in the sense that an expression in a context may be replaced with a different expression yielding the same value, without changing the meaning of the surrounding context. This makes calculations much more straightforward.

Assuming that an imperative program is deterministic, we can say that the final state (σ') is some function of the initial state (σ), say $\sigma' = f(\sigma)$. In functional programming, the program is actually an expression that corresponds to the mathematical function f.

In functional programming, because there are no variables, the concept of state does not exist. The idea of sequence is also meaningless since there is no state to mediate between two commands. However, functional programming presents other advantages such as the use of functions in much more sophisticated ways and the application of recursive mechanisms to simulate repetitive command execution.

Perhaps the main reason for which we prefer functional programs to imperative programs is due to the fact that the first correspond more directly to mathematical objects, and are, consequently, easier to reason. The most appropriate form to assign a meaning to a program is to interpret it as an abstract mathematical object, and this is the aim of *denotational semantics*. This has to be done in an indirect way in imperative programs. A command takes a state and produces another state, so it is associated with a function $\Sigma \to \Sigma$ where Σ is the set of possible values for the state. And this function may also be partial because it can fail to terminate.

Furthermore, in functional programming the non existence of variables and goto statements makes the semantics simpler. Simpler semantics makes reasoning about programs more straightforward. Thus, the possibilities of correctness proofs and transformations increase.

In the functional programming context, the term *lazy evaluation* is applied to indicate that an expression is evaluated only if its value is needed for the program to return its result. Lazy evaluation is used to implement *non-strict functions*. A function is strict when the evaluation of its application starts after the evaluation of its arguments.

In contrast to lazy evaluation, *eager evaluation* is the evaluation of some of all the function arguments that starts before their value is needed. A common example is when a function receives its arguments already evaluated and is applied afterwards, this strategy is also called *call-by-value*.

In agreement with the previous, if evaluating an argument leads to non-termination, and this argument is not needed, eager evaluation will lead to non-termination but lazy evaluation may not.

Lambda Calculus

Because λ-*calculus* is the mathematical foundation of functional programming, it is appropriate to present an overview on this topic. λ-calculus is a formalism proposed by Alonzo Church in 1930 for representing the notion of *computation*, like the *turing machines* proposed by Alan Turing. The basic idea behind the λ-calculus is the interaction study of *functional abstraction* and *function application*.

There are just three kinds of λ-expressions [19] or λ-*terms* in [42]:

- Variables: x, y, z, etc. The functions denoted by variables are determined by the links their variables have to the environment.

- Function applications: If E_1 and E_2 are λ expressions, then $(E_1 E_2)$ denotes the result of applying the function denoted by E_1 to the function denoted by E_2.

- Abstractions: If V is a variable and E is a λ-expression, then $\lambda V.E$ is an abstraction with *bound variable* V and *body* E.

So, the syntax of λ-terms is very simple:

$$E ::= V|E_1E_2|\lambda V.E$$

λ-calculus is a notation for defining functions, and the usual representation for defining functions $f(a) = e$ is reformatted to $f = \lambda a.e$.

In λ-expressions the occurrence of variables can be of two types: *free* or *bound*. Occurrence is free when the variable is not within the scope of a λV, otherwise the variable is bound. So, in the next expression

$$\lambda x.xy$$

x is bound and y is free.

In this formalism the computation process takes place by substitution applying the following rules:

- Renaming bound variables (α-conversion). $\lambda x.E \rightarrow_\alpha \lambda y.E[x/y]$, where $E[x/y]$ is the result of replacing the free occurrences of x with y in E.
- Substitution rule (β-conversion). $\lambda x.E\ F \rightarrow_\beta E[F/x]$, where $E[x/F]$ is the result of replacing the free occurrences of x with F in E.
- η-conversion.$\lambda x.Ex \rightarrow_\eta E$ where the trivial expression $\lambda x.Ex$ collapses down to E.

Some examples of λ-reductions are:

$$(\lambda x.(xx))(xy) \rightarrow_\beta (xy)(xy)$$
$$(((\lambda x.(\lambda y.((add\ x)y)))\underline{3})\underline{4}) \rightarrow_\beta ((\lambda y.((add\ \underline{3})y))\underline{4}) \rightarrow_\beta ((add\ \underline{3})\underline{4})$$

where $\underline{3}$ is a λ-expression which represents natural 3, and *add* is a λ-expression denoting a function satisfying

$$((add\ \underline{m})\underline{n}) = \underline{m} + \underline{n}$$

In λ-calculus all functions are unary, and the functions with multiple arguments are simulated by *carrying*. So a function with two arguments $f(a, b) = e$ is reformatted to $f = \lambda a.\lambda b.e$ expressing that the function first takes one argument and returns a function to one argument that returns the final result. At the same time if the signature of the function was $f : A \times B \longrightarrow C$ it is reformatted to $f : A \longrightarrow (B \longrightarrow C)$. So for example

$$(\lambda x\ y.x + y)1\ 2 = (\lambda y.1 + y)2 = 1 + 2$$

where we can see that the function application is associative to the left.

It is also important to have the difference between *equality* and *identity* clear. Two λ-expressions are equal if they can be transformed into each other by a sequence of (backward or forward) λ-conversions. And two λ-expressions are identical if they consist of exactly the same sequence of characters. So for example, $\lambda x.x$ and $\lambda y.y$ are equal but not identical.

For more details on this topic see [2, 31, 43].

2.3.3 Denotational / Operational Semantics

(say we have decided to stick to the former, why?)
The main aspects of programming languages are the syntax and the semantics. Syntax determines which symbol sequences are permitted phrases of the language, and in this way checks the appearance of well-formed programs in the language [44]. Semantics defines the meanings of sentences in programming languages, explains what phrases mean (denote). So, semantics precisely expresses the computational meaning of constructs applying formal semantics. The computational meaning of a program refers to what really happens when a program is executed on a concrete computer. But expressing this is very complex to describe in full. Instead, semantics represents the relevant features of all possible executions using abstract entities. Important features are usually: input - output relations and whether the execution of a program terminates or not. Nontermination is usually denoted by $\perp$.

In general, a description is formal when it is written in a notation that already has a precise meaning. Like syntax, language semantics can be formalized. But, unfortunately, unlike syntax, which can be formalized via BNF rules, semantics is harder to formalize. Some approaches to formalize semantics are: (structural) *operational semantics*, *axiomatic semantics*, *denotational semantics*. There is also an intermediate method between operational semantics and denotational semantics: *natural semantics*.

Here we shall only make reference to operational semantics and denotational semantics. Operational semantics explicitly describes how programs compute in stepwise fashion. The meaning of a well-formed

program is the trace of the computation steps that results from processing the program input [44]. In other words, the semantics of a program is defined as a sequence of computation steps produced by rewriting (transition) rule schemes.

A drawback of operational semantics is the emphasis it places upon state sequences [44]. On the contrary, denotational semantics defines the meaning of programs based on underlying *semantic domains* (intrinsic meaning, "sets" of mathematical objects). In denotational semantics, a mathematical structure (denotational model) is chosen. So a semantic or *valuation function* maps programs to denotations

$$[\![]\!] : L \longrightarrow V \tag{2.2}$$

where L represents the syntactic terms and V is a set of semantic values. That is, the meaning of the expressions are mathematical objects. Since the valuation functions are defined by structural induction, the semantics can be expressed in a compositional manner. The meaning of the complete program is defined in terms of the program constructs, and each of these in terms of its subcomponents, and so forth. The denotation of a part of a program represents its contribution to the overall behavior [31].

λ-calculus notation is commonly applied to denotational semantic expressions.

From our point of view, we are interested in establishing which is the best formal semantic technique to apply to a reverse engineering strategy like the one we are defining in this thesis. In order to solve this matter, such techniques must be analyzed through some dimensions. First, as we indicated before, semantics alone cannot express everything that happens when a program is executed. Thus, the formal definition technique has to be able to reveal those aspects that are important for the reverse engineering strategy. Second, and due to the fact that we are dealing with slicing techniques, the semantic technique must be capable of supporting some approaches to program analysis such as *control-flow analysis* and *data-flow analysis.* Third, the semantic technique must be easily applicable. Finally, and very related to the last aspect, the semantic technique must be easily tractable since in most of the cases, some transformations will be required before the application of algebraic laws.

Having analyzed both techniques we have arrived to the following conclusions:

- Operational semantics is well-suited for capturing the meaning of the program at a much closer machine

level. In contrast, denotational semantics provides a more abstract meaning which raises the properties that are relevant for us.

- The transition rules of operational semantics are not so easily applicable. On the contrary, modern functional programming languages such as **HASKELL**, **ML**, etc., can implement denotational semantics.
- The transformations that can be executed with denotational semantics are easily executed and well known (*fold, undfold, definition, instantiation, etc.*). In contrast, with operational semantics we do not know what kind of transformations will be necessary for the application of algebraic laws.

Therefore, we conclude that denotational semantics fulfills most of the requirements we need.

2.4 Summary

In this chapter we have developed a set of relevant topics that we consider are important to know to understand this work. Knowledge about the reverse engineering field is necessary since it is the application context of the approach we propose. Specifically, the slicing technique that plays a relevant role in **RE** activities, also has an important role in the formal reverse engineering strategy that we propose. We need to refer to formal methods because the specifications that we recover (calculate) will have an algebraic format. As the semantics we will use subsequently (chapter 5) will be expressed in functional programming, this topic and its underlying formalism needs to be present as well.

Chapter 3

Algebra of Programming

In section (5.1) we have already stated we shall use algebra of programming as the calculus mechanism to generate more abstract specifications. Specifically, we refer to the use of the *reification calculus* based on **Sets** [36, 38] in a reverse way. As we can expect, this technology was originally developed for forward engineering processes [1] [34, 35, 36] *et al.*

The interpretation of programs as algebraic models paves the way to analyze them in two complementary dimensions: data dimension and operation dimension. We have already mentioned that our aim is set on reverse algorithmical structures and establishing the application of set calculus on data in [37, 33]. Consequently, only a brief overview on some fundamental laws will be introduced in this section.

3.1 Basic concepts

The usual notation for functions are

[1]Following the concept given in [7] on forward engineering

$$f : A \longrightarrow B$$

$$A \xrightarrow{f} B$$

$$f : B \longleftarrow A$$

$$B \xleftarrow{f} A$$

all expressing the same: f is a kind of process abstraction which takes A-values and produces B-values. We also say that $A \longrightarrow B$ is the type of the function and that A and B are sets of values, frequently termed *types*.

Two basic functions are the *identity function*

$$\begin{array}{ll} id_A : & A \longrightarrow A \\ id_A\ a & \stackrel{def}{=} a \end{array} \tag{3.1}$$

and the *constant function*

$$\begin{array}{ll} \underline{c} : & A \longrightarrow C \\ \underline{c}\ a & \stackrel{def}{=} c \end{array} \tag{3.2}$$

But there are complex "functional units" implemented in the source code which can involve other simpler functional units. So, it is essential to have at one's disposal a formal mechanism to model those functional units at a higher level of abstraction. The simplest way to combine functions is by means of *composition*. This operator models situations where one function produces a result that is taken by another function, then the other function can be defined: $g \cdot f$.

$$\begin{array}{lcl} g \cdot f & : & C \longleftarrow A \\ (g \cdot f)a & \stackrel{def}{=} & g(f(a)) \end{array} \tag{3.3}$$

where

$$\left\{ \begin{array}{ll} f : & B \longleftarrow A \\ g : & C \longleftarrow B \end{array} \right.$$

3.2 Products toolbox

But there are cases where two functions do not compose, for example $f : B \longleftarrow A$ and $g : C \longleftarrow A$. Here the domain of one is not the codomain of the other, but they share the same domain. Therefore we may construct a new function combinator

$$\begin{array}{lcl} \langle f, g\rangle & : & C \longrightarrow A \times B \\ \langle f, g\rangle c & \stackrel{def}{=} & (f\ c, g\ c) \end{array} \tag{3.4}$$

which is termed *split* or *fork* in [18], and $A \times B$ is termed the *product* of A and B. In [10] the notation $f \triangle g$ expresses the same operator.

To say that $A \times B$ is a product is to say that for $f : A \longleftarrow C$ and $g : B \longleftarrow C$ there is a unique function h such that $\pi_1 \cdot h = f$ and $\pi_2 \cdot h = g$. The uniqueness is expressed by the following *universal property*

$$h = \langle f, g\rangle \quad \Leftrightarrow \quad \left\{ \begin{array}{lcl} \pi_1 \cdot h & = & f \\ \pi_2 \cdot h & = & g \end{array} \right. \tag{3.5}$$

In the same way that the cartesian product $A \times B$ can retrieve A data and B data by means of the *projections* π_1 and π_2, or *exl* and *exr* respectively in [18] (also called *selectors* in [38])

$$A \xleftarrow{\pi_1} A \times B \xrightarrow{\pi_2} B$$

defined by

$$\begin{array}{lcl} \pi_1(a, b) & = & a \\ \pi_2(a, b) & = & b \end{array} \tag{3.6}$$

f and g can also be retrieved from $\langle f, g\rangle$

$$\begin{array}{lcl} \pi_1 \cdot \langle f, g\rangle & = & f \\ \pi_2 \cdot \langle f, g\rangle & = & g \end{array} \tag{3.7}$$

in such a way that one of the functions is omitted. We will refer to this fact as the $\times - cancellation$ property.

But what occurs when the domains are different?

$$\begin{array}{ll} f : & B \longleftarrow A \\ g : & D \longleftarrow C \end{array}$$

Clearly the split combinator does not work any longer. However, we may put together both domains and recover them following (3.6), so that the expression $\langle f \cdot \pi_1, g \cdot \pi_2 \rangle$ is well-typed having domain $A \times C$ and codomain $B \times D$.

We will use $f \times g$ (product f and g) to express this functional combination.

$$\begin{array}{lcl} f \times g & : & A \times C \longrightarrow B \times D \\ f \times g & \stackrel{def}{=} & \langle f \cdot \pi_1, g \cdot \pi_2 \rangle \end{array} \tag{3.8}$$

On the other hand, we can "reason" these functional combinators and think on how to relate some of them. In this way we may put together split and $\cdot$ (composition)

$$\begin{array}{ccccc} A & \xleftarrow{\pi_1} & A \times B & \xrightarrow{\pi_2} & B \\ & {\scriptstyle g} \nwarrow & \uparrow {\scriptstyle \langle g,h \rangle} & \nearrow {\scriptstyle h} & \\ & & C & & \\ & {\scriptstyle g \cdot f} & \uparrow {\scriptstyle f} & {\scriptstyle h \cdot f} & \\ & & D & & \end{array} \qquad \langle g, h \rangle \cdot f = \langle g \cdot f, h \cdot f \rangle \tag{3.9}$$

and we may see that the split is right-distributive with respect to $\cdot$ (composition). This property is known as $\times - fusion$ property. Unfortunately, $f \cdot \langle g, h \rangle$ does not work either, but if $f = i \times j$ we may say that

$$\begin{array}{ccccc} A & \xleftarrow{\pi_1} & A \times B & \xrightarrow{\pi_2} & B \\ {\scriptstyle i} \uparrow & & \uparrow {\scriptstyle i \times j} & & \uparrow {\scriptstyle j} \\ D & \xleftarrow{\pi_1} & D \times E & \xrightarrow{\pi_2} & E \\ & {\scriptstyle g} \nwarrow & \uparrow {\scriptstyle \langle g,h \rangle} & \nearrow {\scriptstyle h} & \\ & & C & & \end{array} \qquad (i \times j) \cdot \langle g, h \rangle = \langle i \cdot g, j \cdot h \rangle \tag{3.10}$$

is the $\times - absortion$ property (see proof in [41]). In this graph we may also see two relations between $\times$ and the projections

$$i \cdot \pi_1 = \pi_1 \cdot (i \times j) \tag{3.11}$$

$$j \cdot \pi_2 = \pi_2 \cdot (i \times j) \tag{3.12}$$

demonstrable following (3.8).

Two special properties are

$$(g \cdot h) \times (i \cdot j) = (g \times i) \cdot (h \times j) \tag{3.13}$$

$$id_A \times id_B = id_{A \times B} \tag{3.14}$$

the first being a kind of "bi-distribution" of $\times$ with respect to $\cdot$ ($\times - functor$ property), and the latter has to do with identity functions($\times - functor - id$ property). Both are termed *functorial* properties.

3.3 Co-products toolbox

On the other hand, we may consider the "dual" situation to the split. So, we have two functions with the same codomain but different domains: $f : C \longleftarrow A$ and $g : C \longleftarrow B$. In this case, we need to define a datatype for the operator to resolve which type to be applied on. $A \cup B$ seems a solution, but if $A \cap B \neq \emptyset$, it is inviable. Therefore it is necessary to define a mechanism that allows us to know where the data come from. The *disjoint union* can overcome this problem

$$A \xrightarrow{i_1} A + B \xleftarrow{i_2} B$$

where two functions are defined

$$\begin{array}{rcl} i_1\ a & = & (t_1, a) \\ i_2\ b & = & (t_2, b) \end{array} \tag{3.15}$$

that individualize each element in $A + B$, the *coproduct* of A and B. On the base of this solution we may define the new operator

$$\begin{array}{rcl} [f, g] & : & A + B \longrightarrow C \\ [f, g]\ x & \stackrel{def}{=} & \left\{ \begin{array}{lcl} x = i_1\ a & \Rightarrow & f\ a \\ x = i_2\ b & \Rightarrow & g\ b \end{array} \right. \end{array} \tag{3.16}$$

called *either* operator (either f or g), and a different notation is applied in [10], $f \bigtriangledown g$.

As in the case of product, to say that $A+B$ is a coproduct is to say that for $f : C \longleftarrow A$ and $g : C \longleftarrow B$ there is a unique function h such that: $h \cdot i_1 = f$ and $h \cdot i_2 = g$. The uniqueness is expressed by the following *universal property*

$$h = [f, g] \quad \Leftrightarrow \quad \begin{cases} h \cdot i_1 & = & f \\ h \cdot i_2 & = & g \end{cases} \tag{3.17}$$

The duality with split operators simplifies reasoning either operators. Thus, we may introduce $f + g$ as the dual of $f \times g$

$$f + g \stackrel{def}{=} [i_1 \cdot f, i_2 \cdot g] \tag{3.18}$$

Next we list the +-properties that express this duality

+-cancellation:

$$\begin{array}{lcl} [f, g] \cdot i_1 & = & f \\ [f, g] \cdot i_2 & = & g \end{array} \tag{3.19}$$

+-reflexion:

$$[i_1, i_2] = id_{A+B} \tag{3.20}$$

+-fusion:

$$f \cdot [g, h] = [f \cdot g, f \cdot h] \tag{3.21}$$

+-absorption:

$$[f, g] \cdot (i + j) = [f \cdot i, g \cdot j] \tag{3.22}$$

+-functor:

$$(f \cdot g) + (i \cdot j) = (f + i) \cdot (g + j) \tag{3.23}$$

+-functor-id:

$$id_A + id_B = id_{A+B} \tag{3.24}$$

Either operator and its dual are related to each other by the *exchange law*. Given the functions $B \xleftarrow{f} A$, $D \xleftarrow{g} A, B \xleftarrow{h} C, D \xleftarrow{k} C$ we may express the function of type $B \times D \longleftarrow A + C$ in two alternative ways

$$[\langle f, g\rangle, \langle h, k\rangle] = \langle [f, h], [g, k]\rangle \tag{3.25}$$

See proof in [17, 18].

An example of a function which is in exchange-law format is *undistr*. Cartesian product distributes over disjoint sum: so

$$A \times (B + C) \cong (A \times B) + (A \times C)$$

are isomorphic objects, as are

$$(B + C) \times A \cong (B \times A) + (C \times A)$$

Therefore we shall have two functions

$$\begin{array}{lcl} undistl & : & A \times (B + C) \longleftarrow (A \times B) + (A \times C) \\ undistl & \stackrel{def}{=} & [id \times i_1, id \times i_2] \\ & \stackrel{def}{=} & \langle [\pi_1, \pi_1], (\pi_2 + \pi_2)\rangle \end{array} \tag{3.26}$$

and

$$distl : (A \times B) + (A \times C) \longleftarrow A \times (B + C) \tag{3.27}$$

but the last is not easy to construct, and we can only assume the existence of a *distl* function and that it is inverse to *undistl*.

3.4 Conditionals

Given a predicate $Bool \xleftarrow{p} A$ we may define its *guard* $A + A \xleftarrow{p?} A$ in *pointwise notation* as follows

$$(p?)a = \begin{cases} pa & \Rightarrow & i_1 a \\ \neg(pa) & \Rightarrow & i_2 a \end{cases} \tag{3.28}$$

So, we may go to *pointfree notation* using "McCarthy Conditional"

$$p \longrightarrow f, g$$

defining it as

$$p \longrightarrow f, g \quad \stackrel{def}{=} \quad [f, g] \cdot p? \tag{3.29}$$

assuming that $B \xleftarrow{f} A, B \xleftarrow{g} A$.

One of the best known properties derived from (3.29), e.g *McCarthy conditional fusion law* is

$$h \cdot (p \longrightarrow f, g) \quad = \quad p \longrightarrow h \cdot f, h \cdot g \tag{3.30}$$

or in pointwise notation

$$h \cdot (if\ p\ then\ f\ else\ g) = if\ p\ then\ h \cdot f\ else\ h \cdot g$$

which is a consequence of (3.21).

But we may also have other equivalent expressions

$$(if\ p\ then\ f\ else\ g) \cdot h = if\ p \cdot h\ then\ f \cdot h\ else\ g \cdot h$$

provable as follows. In pointfree notation

$$[f, g] \cdot p? \cdot h = [f \cdot h, g \cdot h] \cdot (p \cdot h)?$$

starting from the right hand side

$$\begin{aligned}
&= \quad if\ (p \cdot h)\ then\ (f \cdot h)\ else (g \cdot h) \\
&\qquad \{McCarthy's\ conditional\} \\
&= \quad [f \cdot h, g \cdot h] \cdot (p \cdot h)? \\
&\qquad \{By\ (3.22)\}
\end{aligned}$$

$$= \quad [f,g] \cdot (h+h) \cdot (p \cdot h)?$$

$$\{theorem\ 16\ in\ [17]\}$$

$$= \quad [f,g] \cdot p? \cdot h$$

$$\{associativity\}$$

$$= \quad ([f,g] \cdot p?) \cdot h$$

$$\{McCarthy's\ conditional\ again\}$$

$$= \quad (if\ p\ then\ f\ else\ g) \cdot h$$

a shorter path

$$\begin{aligned} [f,g] \cdot p? \cdot h \quad &= \\ &\quad \{By\ theorem\ 16\ in\ [17]\} \\ &= [f,g] \cdot (h+h) \cdot (p \cdot h)? \\ &\quad \{By\ (3.22)\} \\ &= [f \cdot h, g \cdot h] \cdot (p \cdot h)? \end{aligned}$$

We may also have

$$if\ not \cdot p\ then\ f\ else\ g = if\ p\ then\ g\ else\ f$$

in pointfree notation

$$[f,g] \cdot (\neg \cdot p)? = [g,f] \cdot p?$$

beginning from the right hand side

$$\begin{aligned} [f,g] \cdot (\neg \cdot p)? \quad &= \\ &\quad \{By\ theorem\ 18\ in\ [17]\} \\ &= [f,g] \cdot swap \cdot p? \\ &\quad \{swap = [i_2, i_1]\} \\ &= [f,g] \cdot [i_2, i_1] \cdot p? \\ &\quad \{By\ (3.22)\} \\ &= [[f,g] \cdot i_2, [f,g] \cdot i_1] \cdot p? \\ &\quad \{By\ (3.19)\} \\ &= [g,f] \cdot p? \end{aligned}$$

Other two easy to prove expressions are

$$if\ p\ then\ f\ else\ f = f$$

$$if\ True\ then\ f\ else\ g = f$$

both in pointfree notation

$$\begin{array}{lcl} [f, f] \cdot p? & = & f \\ [f, g] \cdot \underline{True}? & = & f \end{array}$$

respectively, where $\underline{True}$ is a constant function (3.2).

3.5 Recursion

The calculus on pair of datatypes does not have sufficient expressive power to model more complex structures. The real expressive power lies on *recursive datatypes* [18]. The next expression is a typical recursive definition

$$L \cong 1 + A \times L \tag{3.31}$$

because L is present in its own definition, and where $\cong$-symbol is the *set-theoretical isomorphism* [41]. Since (3.31) is isomorphic it must have a function

$$in : L \longleftarrow 1 + A \times L \tag{3.32}$$

that transforms $1 + A \times L$ datatypes into L datatypes; and another function

$$out : L \longrightarrow 1 + A \times L \tag{3.33}$$

that transforms L datatypes into $1 + A \times L$ datatypes.

According to (3.16) we may define in as $in = [in_1, in_2]$, where $in_1 = \underline{[\,]}$ is the "nil pointer" and $in_2 = cons$ (the "left append" sequence constructor)

$$\begin{array}{lcl} cons & : & A \times A^* \longrightarrow A^* \\ cons(a, [a_1, \ldots, a_n]) & = & [a, a_1, \ldots, a_n] \end{array} \tag{3.34}$$

and A^* is a solution for $L = 1 + A \times L$ from which (3.31) is generated. Besides, applying (3.3) and (3.18) we may define the out function as follows

$$out \stackrel{def}{=} (\langle\rangle + \langle hd, tl\rangle) \cdot (=_{[\,]}?) \tag{3.35}$$

Obviously in and out are each other's inverses, so $in \cdot out = id$ is respected. See proof in [41] et al.

Going back to (3.31), we can provide many solutions and any datatype X *inductively* defined by means of a constant and a binary constructor accepting A and X as parameters will be sufficient. These solutions at abstract level are all the same since they all capture the same idea: the abstract notion of a list of symbols.

In general, we can say that each function of the form

$$B \xleftarrow{f} A^* \tag{3.36}$$

can be written following a recursive scheme. In this scheme we have a $k \in B$ (a constant) and $B \xleftarrow{g} A \times B$ (a binary constructor). Two typical examples are:

- The function that adds the elements of a list so that $k = 0$ and $add = x + y$.
- The function that multiplies the elements of a list so that $k = 1$ and $mult = xy$.

Each instance of (3.36) is determined by the pair (k, g) in the inductive datatype domain. This pair is called $algebra$ and it can be packed following (3.16)

$$B \xleftarrow{[\underline{k}, g]} 1 + A \times B \tag{3.37}$$

Replacing L in (3.32) with the solution A^* we integrate this function with (3.37) on the same diagram

$$\begin{array}{ccc} A^* & \xleftarrow{in} & 1 + A \times A^* \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle id+id\times f} \\ B & \xleftarrow[{[\underline{k}, g]}]{} & 1 + A \times B \end{array} \tag{3.38}$$

where the following property is revealed

$$f \cdot in = [\underline{k}, g] \cdot (id + id \times f) \tag{3.39}$$

which can be written in a simpler way

$$f \cdot in = \alpha \cdot F\ f \tag{3.40}$$

having $\alpha = [\underline{k}, g]$ and $F\ f = id + id \times f$. Thus, providing α, we may define f. This dependency of f on α is expressed as

$$(\!|\alpha|\!) \cdot in = \alpha \cdot F\ (\!|\alpha|\!) \tag{3.41}$$

and we read it as *α-catamorphism* instead of f. This concept is called *fold* in [18] et. al. It is necessary to emphasize that two important concepts are applied in (3.40) and (3.41):

- **Functor**: An $(endo) - functor$ **F** is an operation on the objects and arrows of a category:
  - if A is an object of $\mathcal{C}$ category then $F\ A$ also is
  - if f is an arrow of $\mathcal{C}$ category then $F\ f$ also is

  and the next properties are met
  - identity: $F(id_A) = id_{FA}$
  - composition: $F(f \cdot g) = (F\ f) \cdot (F\ g)$
- **F-Algebra:** Any structure $(A,\ A \xleftarrow{\alpha} F\ A\ )$ is an F-algebra. So for example $(A^*,\ A^* \xleftarrow{[\underline{Nil}, cons]} 1 + A \times A^*\ )$ is a F_{A^*}-algebra.

As we have already indicated, A^* is not the only solution for (3.31). Another solution can be obtained inductively defining a datatype X by means of a constant and a binary constructor accepting A and X as parameters [41]. In other words, we are referring to *monomorphic datatypes* [18]. Therefore, in order to obtain an abstract definition for all datatype declarations with the previous properties, we need to consider a datatype T to which only one constructor is associated

$$in_T : F\ T \longrightarrow T \tag{3.42}$$

where F is a functor: $F\ T = 1 + A \times T$.

3.6 Anas, catas and hylomorphisms: An overview

We must also notice that in (3.38) if we replace *in* with *out* we get an equivalent diagram. The dual of this last diagram can be obtained redefining (3.36) to $B \xrightarrow{f} T$ (where we have replaced A^* with the more general T). Thus we have

$$\begin{array}{ccc} T & \xrightarrow{out} & 1 + A \times T \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle id+id\times f} \\ B & \xleftarrow[{[\underline{k},g]}]{} & 1 + A \times B \end{array} \qquad \begin{array}{ccc} T & \xleftarrow{in} & 1 + A \times T \\ {\scriptstyle f}\uparrow & & \uparrow{\scriptstyle id+id\times f} \\ B & \xrightarrow[ana]{} & 1 + A \times B \end{array} \tag{3.43}$$

and as we express in (3.41) $f = (\![\underline{k}, g]\!)$ with respect to the left hand side diagram and $f = [(ana)]$ with respect to the right hand side diagram. We read $[(ana)]$ as the T*-anamorphism* (*unfold* in [18] et. al) induced by ana which is defined by

$$[(ana)] = in \cdot (id + id \times [(ana)]) \cdot ana \tag{3.44}$$

From (3.43) we can observe that T can act as an intermediate data-structure between T-catamorphism and T-anamorphism. So the next diagram can be constructed

$$\begin{array}{ccc} B & \xleftarrow{[\underline{k},g]} & 1 + A \times B \\ {\scriptstyle (\![\underline{k},g]\!)}\uparrow & & \uparrow{\scriptstyle id+id\times(\![\underline{k},g]\!)} \\ T & \xleftarrow{in} & 1 + A \times T \\ {\scriptstyle [(ana)]}\uparrow & & \uparrow{\scriptstyle id+id\times[(ana)]} \\ C & \xrightarrow[ana]{} & 1 + A \times C \end{array} \tag{3.45}$$

where we observe an evident composition between T-catamorphism and T-anamorphism thus giving rise to the next property

$$(\![\underline{k}, g]\!) \cdot [(ana)] = [\underline{k}, g] \cdot (id + id \times (\![\underline{k}, g]\!)) \cdot (id + id \times [(ana)]) \cdot ana$$

which after some calculations is tranformed into

$$(\![\underline{k}, g]\!) \cdot [(ana)] = (id + id \times (\![\underline{k}, g]\!) \cdot [(ana)]) \cdot ana \tag{3.46}$$

where $B \xleftarrow{(\![\underline{k},g]\!)\cdot[(ana)]} C$ is defined.

In this way we can construct functions composed by other two functions, one producer and the other consumer. The first function (anamorphism) generates a value of the intermediate datatype from the input argument, as long as the second function (catamorphism) reduces the intermediate data-structure and calculates the final result.

The composition $(\![\underline{k}, g]\!) \cdot [(ana)]$ is called A^**-hylomorphism* and is denoted by $[\![[\underline{k}, g], ana]\!]$

$$[\![[\underline{k}, g], ana]\!] = (\![\underline{k}, g]\!) \cdot [(ana)] \tag{3.47}$$

Below, we develop a brief exercise in order to get a better understanding on the previous concepts. A way of computing n^2 is to add the n first odd numbers. For example $4^2 = 1 + 3 + 5 + 7$. Based on this calculus mechanism we express the function

$$sq\ n \stackrel{def}{=} n^2$$

From the right diagram in (3.43), where we replace A with $\mathcal{N}$, we know that

$$f = in \cdot (id + id \times f) \cdot [(ana)] \tag{3.48}$$

which has the same structure as (3.46). We also know that (3.48) is equivalent to

$$f = [\underline{Nil}, cons \cdot (id \times f)] \cdot [(ana)] \tag{3.49}$$

Because we need to distinguish the input 0 from all the others $f\ 0 = Nil$ will hold. We decompose $[(ana)]$ as follows

$$\begin{array}{ccc} N & \xrightarrow{=_0?} & N + N \\ & \searrow^{[(ana)]} & \downarrow^{\langle\rangle + h} \\ & & 1 + N \times N \end{array} \tag{3.50}$$

where $=_0?$ is defined by (3.28) and $\langle\rangle : 1 \longleftarrow N$. What happens with h? Remembering (3.4) and (3.18) h will have the form $\langle _, _ \rangle$. Its first component will provide the element with which *cons* will construct the output, while its second component will be the argument for the next recursive call. As we are interested in the generation of first n odd numbers, we define $h = \langle odd, pred \rangle$ where

$$odd \stackrel{def}{=} 2 * n - 1 \tag{3.51}$$

computes the nth odd number and

$$pred \stackrel{def}{=} n - 1 \tag{3.52}$$

computes the predecessor of n. Thus, putting the previous components together we have

$$[(ana)] = (\langle\rangle + \langle odd, pred \rangle) \cdot (=_0?) \tag{3.53}$$

In this way we obtain the generator or producer for (3.47). Besides, we need a consumer that adds the elements of the previously generated structure. That is, we need a function of type $N \longleftarrow 1 + N \times N$. By (3.16) we know that this function will have the form $[_, _]$. Specifically, the first component of type $N \longleftarrow 1$ will mean the constant function always giving back 0. This function is denoted by $\underline{0}$. Whereas the second component of type $N \longleftarrow N \times N$ will be the well-known *add* function

$$add(x, y) \stackrel{def}{=} x + y \tag{3.54}$$

So we get

$$N \xleftarrow{[\underline{0}, add]} 1 + N \times N \tag{3.55}$$

Replacing the components obtained in (3.48) we have

$$\begin{array}{rcl} f & = & [\underline{0}, add] \cdot (id + id \times f) \cdot (\langle\rangle + \langle odd, pred\rangle) \cdot (=_0?)) \\ & & by\ (3.23)\ and\ (3.1) \\ & = & [\underline{0}, add] \cdot (\langle\rangle + (id \times f) \cdot \langle odd, pred\rangle) \cdot (=_0?)) \\ & & by\ (3.10)\ and\ (3.1) \\ & = & [\underline{0}, add] \cdot (\langle\rangle + \langle odd, f \cdot pred\rangle) \cdot (=_0?)) \\ & & by\ (3.22)\ and\ (3.2) \\ & = & [\underline{0}, add \cdot \langle odd, f \cdot pred\rangle] \cdot (=_0?)) \\ & & by\ (3.29) \\ & = & (=_0?) \longrightarrow \underline{0}, add \cdot \langle id, f \cdot pred\rangle \end{array}$$

This last expression is expressed in pointwise notation as

$$\begin{array}{rcl} f\ 0 & = & [\underline{0}, add \cdot \langle odd, f \cdot pred\rangle](i_1 0) \\ & & by\ (3.19) \\ & = & \underline{0}\ 0 = 0 \end{array}$$

and

$$\begin{array}{rcl} f\ n & = & [\underline{0}, add \cdot \langle odd, f \cdot pred\rangle](i_2 n) \\ & & by\ (3.19) \\ & = & add \cdot \langle odd, f \cdot pred\rangle\ n \\ & & by\ (3.4), (3.54), (3.51), and\ (3.52) \\ & = & (2 * n - 1) + f(n-1) \end{array}$$

In summary

$$\left\{ \begin{array}{rcl} f\ 0 & = & 0 \\ f\ n & = & (2 * n - 1) + f(n-1) \end{array} \right.$$

that, in VDM-SL

```
sq : nat -> nat

sq(n) ==

 if n=0

 then 0

 else (2*n-1)+sq(n-1)
```

We end this section presenting an important law where the recently introduced concept of catamorphism is applied. Let us suppose that we have two functions on the same inductive datatype: $f : A \longleftarrow T$ and $g : B \longleftarrow T$. So, the next two commutative diagrams can be constructed

$$\begin{array}{ccc} T & \xleftarrow{in} & F\,T \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle F\langle f,g\rangle} \\ B & \xleftarrow[h]{} & F(A\times B) \end{array} \qquad \begin{array}{ccc} T & \xleftarrow{in} & F\,T \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle F\langle f,g\rangle} \\ B & \xleftarrow[k]{} & F(A\times B) \end{array} \tag{3.56}$$

where we have to resort to (3.4) to express that both f and g operate on the same datatype T. Combining these two diagrams we have

$$\begin{array}{ccc} T & \xleftarrow{in} & F\,T \\ {\scriptstyle \langle f,g\rangle}\downarrow & & \downarrow{\scriptstyle F\langle f,g\rangle} \\ A\times B & \xleftarrow[\langle h,k\rangle]{} & F(A\times B) \end{array} \tag{3.57}$$

From here, we can extract the next property

$$\left\{ \begin{array}{lcl} f\cdot in & = & h\cdot F\langle f,g\rangle \\ g\cdot in & = & k\cdot F\langle f,g\rangle \end{array} \right. \equiv \langle f,g\rangle = (\!|\langle h,k\rangle|\!) \tag{3.58}$$

which is called *mutual recursion law* or “Fokkinga law”.

3.7 Monads

Need for program calculi, cf analogy with Laplace transform etc

Chapter 4

Formal Reverse Engineering

4.1 Approaches to Formal Reverse Engineering

In this section, two strategies for applying formal methods to **REP** are presented. We consider these proposals are the most serious intents to provide a mathematical base to **REP**.

4.1.1 Approach I

Among the intents for the application of formal methods to **REP** we emphasize [12, 6, 14, 15, 13]. This approach uses the logical properties of programs to abstract formal specification.

Preconditions and *postcondition* are used to define the states of the program, the first describe the initial state and the latter the final state. A *weakest precondition wp(S,R)* is also defined: the set of states in which the statement S can begin execution and terminate with postcondition R true.

For each programming construct (assignments, alternatives, iteratives, etc) pre and postconditions are defined. So, for example, for the statement **x:=e** an annotated code is constructed

$$\begin{array}{l} \{pre\} \\ x := e \\ \{x = e \wedge pre\} \end{array}$$

And the wp of an assignment statement is expressed as $wp(x := e, R) = R^x_e$ which represents the postcondition R where every occurrence of x is replaced with e.

In agreement with [6], formal specification based on predicate logic have properties that are adequate for the semi-automated identification of objects.

A slight variation on this approach was introduced in [14]. Instead of using the predicate transformer as a guideline for constructing formal specification (wp), a predicate transformer is directly applied to the program (*strongest postcondition, sp*).

Notation by Hoare is applied. So, $Q\{S\}R$ expresses a *partial correctness model of execution*, where given that a logical condition Q holds, if the execution of S terminates, then the logical condition R will hold. Besides, $\{Q\}S\{R\}$ represents a *total correctness model of execution*, where, if the condition Q holds, then S is guaranteed to terminate with condition R true.

New concepts are also introduced. The *weakest liberal precondition wlp(S,R)* defines the set of states in which S can begin the execution and establish R as true if S terminates. The *strongest postcondition sp(S,Q)* predicate transformer indicates that if Q holds, the execution of S results in $sp(S,Q)$ true if S terminates. In this way, $wlp(S,R)$ and $sp(S,Q)$ assume both partial correctness.

Finally a relationship between wlp and sp is established

$$\begin{aligned} Q &\Rightarrow wlp(S,R) \\ sp(S,Q) &\Rightarrow R \end{aligned}$$

This relationship provides a formal basis for translating programming statements into formal specifications, and the symmetry of wlp and sp provides a method for verifying the correctness of the reverse engineering process applying sp.

In **REP** the use of wp involves a known postcondition R. But in **RE** the purpose is to find R; wp can only be used as a guideline to perform **RE**. On the contrary, the use of sp assumes a known precondition Q and that a postcondition will be derived by the direct application of sp. Therefore, the use of sp in **REP** is more appropriate. The strongest postcondition semantics of the Dijkstra guarded command language can be found in [15, 13, 14].

Nevertheless, a new improvement is introduced in [15] via [13, 14]. The specifications constructed by the approach above mentioned, contain a significant amount of algorithmic and implementation detail, these specifications are called *as-built specifications*. As-built formal specifications are generalized following *abstraction match*.

A match is an abstraction match if $i \preceq l$, so that

$$(l_{pre} \rightarrow i_{pre}) \wedge (i_{post} \rightarrow l_{post})$$

l is more abstract than i, or in other words, i is a refinement of l. Given a pre and postcondition specification I with precondition I_{pre} and postcondition I_{post}, the purpose is to obtain an axiomatic specification A such that $I \preceq A$. This can be done modifying I to produce a specification I' that satisfies $I \preceq I'$.

But how do we get I'? Either by strengthening I_{pre}, weakening I_{post}, or both, a specification I' that satisfies $I \preceq I'$ is produced. Different techniques can be applied to weaken the postcondition in agreement with [13]: by deletion of a conjunct, addition of a disjunct, conjunction to implication or by disjunction transformations. And to strengthen the precondition: add a conjunct and delete a disjunct.

Initially, the strategy is to search a specification library looking for those components that satisfy a specific match criterion. In this way, if the query based on as-built specifications is formulated, then the library specification is a generalization of the as-built specification. When a search process fails, a derivation process is executed to obtain an abstraction. The derivation by strengthening the preconditions, weakening the postcondition or both is carried out.

However, because the abstraction obtained satisfies the partial order property of the abstraction match operator, an enormous amount of specifications can be obtained. A tool called SPECGEN is used to handle the complexity of this situation.

In [15] a deeper example is shown where this approach is combined with informal techniques.

4.1.2 Approach II

The DPhil. thesis by Martin Ward [55] introduced a strategy for specification refinement applying a Wide Spectrum Language (**W**SL). The basic idea is to begin from high level descriptions of the program and through successive transformations reach a computable version of the program [29, 30] et al. The complete transformation process should be carried out on the same support: **W**SL. So **W**SL it is not a particular notation but a general language with the capacity to handle expressions at different levels of abstraction.

In [55] the author says that the approach can be also applied in opposite direction, that is, starting

from the source code (generated or not with this strategy) to go up the levels of abstraction until obtaining abstract specifications. It is a *transformational style*: "A *program transformation* is an operation which modifies a program into a different form which has the same external behavior (it is equivalent under a precisely-defined denotational semantics)".

So programs and specifications are parts of the same language, and thus, transformations can be used to demonstrate that a given program is a correct implementation of a given specification [51].

In [46] more details about this **REP** are given. Basically the proposed phases are:

- Translating the program into **W**SL.
- Restructuring transformations.
- Determining suitable higher-level data representations and control structures (Abstract data types).
- Applying transformations to go up the level of abstraction of the data representations and control structures (Restructure and simplify).
- Acceptation test.

The translation of the source code into **W**SL notation, will usually be an automatic process. But in many cases the semantics of a source code may depend on the specific compiler and target machine that execute it. In those situations, the translator will have to omit fewer details, and the redundant details will be removed in future transformations.

The purpose of the re-structuration process is to improve a poor design resulting of subsequent patching and "enhancements". Some of the activities executed are:

- Removal: goto statements, redundant tests, flags, etc..
- Moving: some variables are moved closer to where they are set.
- Merging: for example identical codes.
- Changing structure: unstructured cyclical sentences are replaced by loops, etc.

These activities are carried out by the analyst following heuristics and supported by automatic means. Some of the tools applied are a structure editor, a browser and pretty-printer, a transformation engine and a library of proven transformations, and a collection of translators [50, 51]. This set of tools is called FermaT, an evolution of *Maintainer's Assistant* [46] and *ReForm Tool* [53].

Some examples of transformation are:

- Expand IF statements.

$$\mathbf{if\ B\ then\ S_1\ else\ S_2\ fi;\ S}$$
$$\downarrow$$
$$\mathbf{if\ B\ then\ S_1; S\ else\ S_2; S\ fi}$$

- Loop inversion.

$$\mathbf{do\ S_1; S_2\ od}$$
$$\downarrow$$
$$\mathbf{S_1;\ do\ S_2; S_1\ od}$$

  where **do** . . . **od** is an infinite loop (unbound loop) which can only be terminated by executing an **exit** statement.

These transformations are implemented in the library of proven transformations. In this way the analyst, based on context information, proposes the transformation to perform and the tool to verify whether that transformation is feasible under the current conditions. Once the applicability condition is verified, the system carries out the transformation automatically. Therefore each transformations is proved correct appealing to already proven transformations [52]. But in some cases this is not possible, so translating the kernel language and directly applying the proof techniques will be necessary.

After the execution of these transformations, the true structure of the source code is revealed, obtaining an easier to understand and modify version. The next phase takes as input the generated version and moves up to a higher level of abstraction. To do this, the source code is explored looking for a suitable control and data abstraction.

Because the strategy is based on semantic equivalence, a kernel language with denotational semantics is defined, and definitional extensions are permitted in terms of these basic constructs . The primitive sentences in the Kernel level are [52]:

- Assertion: $\{\mathbf{P}\}$. If formula $\mathbf{P}$ is true, the statement terminates immediately without changing any variable, otherwise it aborts. So for example, if the assertion is at the beginning of the program it can be treated as precondition.
- Guard: $[\mathbf{Q}]$. It always terminates, and enforces $\mathbf{Q}$ to be true at this point in the program without changing the values of any variables.
- Add variables: $\mathbf{add(x)}$. Adds the variables in $\mathbf{x}$ to the state space (if they are not already present) and assigns them arbitrary values. Usually, the arbitrary values may be restricted by a subsequence guard.
- Remove variables: $\mathbf{remove(y)}$. Removes the variables in $\mathbf{y}$ from state space (if they are present).

Compound statements:

- Sequence: $\mathbf{S_1; S_2}$. First executes S_1 then S_2.
- Nondeterministic choice: $\mathbf{S_1} \sqcap \mathbf{S_2}$. Chooses S_1 or S_2 nondeterministically.
- Recursion: $\mu\mathbf{X}.\mathbf{S_1}$. This represents recursive calls to the procedure whose body is $\mathbf{S_1}$, and $\mathbf{X}$ is a statement variable.

At this phase, knowledge of the syntax and semantics of the more abstract statements (specification statements) of **WSL** is also required since this will be the support of the new representations. Some notations applied at this moment include:

- Sequences: $s = \langle a_1, \ldots, a_n \rangle$ where the ith element is denoted by $a[i]$, as long as the subsequence $\langle s_i, \ldots, s_j \rangle$ is denoted by $s[i..j]$, where $s[i..j] = \langle\rangle$ if $i > j$.
- Concatenation: $s \# t = \langle s[1], \ldots, s[l(s)], t[1], \ldots, t[l(t)] \rangle$. Where $l(s)$ is the length of the sequence s.
- Sets: For the sequence s, $set(s) = \{s[i] | \forall i : 1 \leq i \leq l(s)\}$. And there are also the usual operations: $\cup, \cap, \subseteq, \in, \wp(powerset)$.
- Relations and functions: A relation is a subset of $A \times B = \{(a, b) | a \in A \wedge b \in B\}$. A relation f is a function if $\forall x, y_1, y_2.((f(x, y_1) \in f \wedge f(x, y_2) \in f) \Rightarrow y_1 = y_2)$.

- Currying: If $\oplus$ is a binary operator and a and b are values, then $(\oplus)$, $(a\oplus)$ and $(\oplus b)$ are functions where $(\oplus)a = a\oplus$, $(a\oplus)y = a \oplus y$ and $(\oplus b)x = x \oplus b$.

- Map: This operator $(*)$ returns the sequence obtained by applying a given function to each element of a given sequence: $f * \langle a_1, a_2, \ldots a_n\rangle = \langle f(a_1),$
$f(a_2), \ldots, f(a_n)\rangle$.

- And others.

These concepts are the basis of current functional programming and an instructive introduction on them can be found in [16].

REP continues replacing data structures with more abstract ones. So arrays can be replaced by sequences, concrete variables by ghost variables and so on. Because the data representation has changed the control structure must do so as well. *Invariants* are designed to ensure the correctness of the translation [46].

The specification statement is an example of non-executable operations in **W**SL

$$\langle x_1, x_2, \ldots, x_n\rangle = \langle x'_1, x'_2, \ldots, x'_n\rangle.Q$$

that is simplified by $x := x'.Q$ in [52], and means that new values are assigned to the variables in x so that Q is true. Formally

$$x := x'.Q =_{DF} (\{\exists x'.Q\}; (add(x'); ([Q]; (add(x'); ([x = x']; remove(x'))))))$$

But [53] argues that it is very difficult to construct invariants for all the loops in programs which were not developed using modern programming methods. To overcome this problem the proof methods based on weakest preconditions expressed in infinitary logic are proposed. Thus [53] introduces a variant in the proof method applied, going from invariant condition to weakest preconditions in the case of loop constructs.

According to [53] a program (a sequence of formal symbols) can be interpreted as:

- A function from structures to state transformations.

- Given any formula **R** (a condition on the final state) we can construct the formula **WP(S,R)**, the *weakest precondition* of **S** on **R**. This condition specifies the initial state such that program **S** is guaranteed to terminate in a state satisfying **R** if it is started in a state satisfying **WP(S,R)**.

Based on these interpretations, two different notions of refinement can be recognized: *semantic refinement* and *proof-theoretic refinement.*

Semantic Refinement

A *state* is a function which maps a finite and non-empty set V of variables to a set D of values. A state transformation f maps each initial state s on one state space, to the set of possible final states $f(s)$ which may be on a different state space.

A state transformation f is a refinement of a state transformation g if they both have the same initial state and $f(s) \subseteq g(s)$ for every initial state s. Thus, if g is refined by f we write $g \leq f$.

A *structure* for a logical language $\mathcal{L}$ consists on a set of values, plus a mapping between constant symbols, function symbols and relation symbols of $\mathcal{L}$ and elements, functions and relations on the set of values. A *model* for a set of sentences is a structure for the language such that each of the sentences is interpreted as true. In other words, a *model* for a set of sentences is an interpretation of the constants, function symbols and relation symbols of the logical formulae used in the program as elements, functions and relations on a given set of values. So, a model provides an interpretation of programs as state transformations. With such a model, we can calculate to give a true value to a formula giving the true values of its free variables. If the interpretation of S_2 under the structure M refines the interpretation of S_1 under the same structure, then we write $S_1 \leq_M S_2$. If it is true for every model of countable set Δ of sentences of $\mathcal{L}$, then we write $\Delta \models S_1 \leq S_2$.

es = variables
heir value in the
ence?}
mulae with no
}

Proof-Theoretic Refinement

Let S_1 and S_2 be sstaments and R a formula, so we have two formulae in infinitary logic $WP(S_1, R)$ and $WP(S_2, R)$. We can define a refinement relation as follows: S_1 is refined by S_2 if only if the formula $WP(S_1, R) \Rightarrow WP(S_2, R)$ can be proved for every formula R.

According to [52] there are two postconditions which completely characterise the refinement relation. We can define a refinement relation using weakest preconditions on these two postconditions: Let x be a sequence of all the variables assigned to either S_1 or S_2 and let x' be a sequence of new variables the same length as x. If the formulae

$$WP(S_1, x \neq x') \Rightarrow WP(S_2, x \neq x') \text{ } and \text{ } WP(S_1, true) \Rightarrow WP(S_2, true)$$

are provable from the set Δ, then S_1 is refined by S_2: $\Delta \vdash S_1 \leq S_2$.

{Δ set of s
tary first o

So, semantic refinement and proof-theoretic refinement are the same:

Theorem: For any statement S_1 and S_2, and any countable set Δ of sentences of $\mathcal{L}$:

$$\Delta \models S_1 \leq S_2 \Leftrightarrow \Delta \vdash S_1 \leq S_2$$

The next example shows how to apply weakest preconditions to prove the validity of transformations, it is necessary to translate the sentence into the notation to carry out the proof:

$$\Delta \vdash if\ B\ then\ S_1\ else\ S_2\ \approx if\ \neg B\ then\ S_2\ else\ S_1$$

so we have

$$\begin{aligned} WP(if\ B\ then\ S_1\ else\ S_2) &= \\ & \{Representing\ it\ in\ kernel\ notation\} \\ &= WP((([B]; S_1) \sqcap ([\neg B]; S_2)), R) \end{aligned} \tag{4.1}$$

$$= WP(([B]; S_1), R) \wedge WP(([\neg B]; S_2), R) \tag{4.2}$$

$$\begin{aligned} &= B \Rightarrow WP(S_1, R) \wedge \neg B \Rightarrow WP(S_2, R) \Leftrightarrow \\ & (\neg B) \Rightarrow WP(S_2, R) \wedge \neg(\neg B) \Rightarrow WP(S_1, R) \end{aligned} \tag{4.3}$$

$$= WP(([\neg B]; S_2), R) \wedge WP(([B]; S_1), R) \tag{4.4}$$

$$= WP((([\neg B]; S_2) \sqcap ([B]; S_1)), R) \tag{4.5}$$

$$= WP(if\ \neg B\ then\ S_2\ else\ S_1, R)$$

where the definitions applied are the following

(4.1)	$WP(S_1 \sqcap S_2), R) =_{DF} WP(S_1, R) \wedge WP(S_2, R)$
(4.2)	$WP([Q], R) =_{DF} Q \Rightarrow R$
(4.3)	reversing 4.2
(4.4)	reversing 4.1
(4.5)	reversing nondeterministic choice

For application cases of this strategy see [47, 49, 48, 54] et al.

Formal definitions of different slicing methods

Should VDM-SL be used as target: formal definitions above can be delivered in VDM-SL notation.
Examples: what is a slice? what is a conditional slice?
Advantages: type checking!! + material paving the way to a formal specification of a slicer + animation of this slicer + instantiating of this slicer for VDM++

4.2 Analysis

In general, the second of the two strategies is the one with more mathematical foundation. The first approach has some drawbacks, and beyond them is the hidden fact that some transformations are performed in an intuitive manner. In particular:

- It is not clear under which conditions to try weakening the postconditions, strengthening the preconditions, or both. Perhaps the analyst, observing these expressions and based on his own experience and knowledge of the source code, makes the decision.
- The selection of the technique, to weaken the postcondition for example, is intuitive as well.

- No mathematical law proves that the elimination of some components of the specifications is correct, nor that the expressions obtained are correct specifications.

Thus, we cannot say that the process is completely formal.

Regarding the second approach, we have also found some drawbacks that are, in general, application difficulties:

- The analysis process is carried out on each line of the source code. It makes **REP** very complicated to be performed on real programs.

- The structural transformation process is founded on informal information. The maintainer (analyst) uses high-level information, including hints gained from comments, variable names and other documentation to guide the system in its selection of transformation.

- It is necessary to guess the abstraction of the data items and then construct the invariants that relate them.

- During the critical process of abstraction, it is also necessary to guess the specification and then to apply theorems to prove that the original representation is a refinement of the guessed specification.

4.3 Summary

In the strategy that will be presented later on, we try to solve these problems defining a homogeneous strategy. Each executed phase will be supported by the same mathematical mechanism, avoiding intuitive execution of tasks. This also includes the supposition of the next specification and its subsequent proof. Our intention is to calculate the next specifications following some law accessible through calculus.

To overcome the difficulty that involves analyzing the complete source code line by line, we use slicing techniques. This improves the calculus process due not only to diminishing the volume of lines to analyze, but also to allowing to lead the analysis process toward a specific functionality. We can also find the mathematical mechanisms for integrating the specifications obtained from each slice in the calculus applied.

An advantage of our strategy with regard to the second approach is that the analysis process is simplified by slicing techniques. The fragmentation of the source code in simpler units makes the **REP** process easier

to apply. It is clear that our approach has the formal mechanism to combine the partial specifications to re-construct the complete specification of the program.

Chapter 5

Formalization of a Strategy for Reverse Engineering

5.1 Introduction

In this chapter we shall develop the conceptual components that integrate the approach we are proposing. The strategy of formal **REP** we are going to define takes a different approach to those described in chapter 4. First, the source code is decomposed in smaller parts (slices) applying slicing techniques. Starting from here, the semantics of each slice is expressed in **HASKELL** or **VDM-SL** . Perhaps some transformations will be necessary before the semantics can be handled by some algebraic law.

In order to understand the case studies developed in this chapter, the reader will have to "look askance" the laws presented in chapter 3.

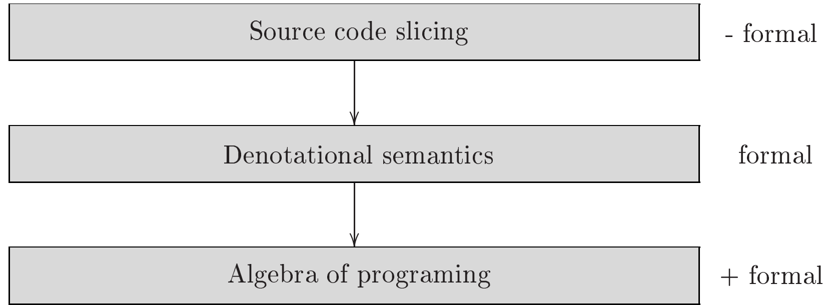
5.2 Scope

Choice of scope: Haskell? perhaps choose VDM-SL target notation, cf ISO/IEC 13817-1 standard specification language.

In the current work we have only considered source codes written in imperative language. Further research will have to be done in order to take the current formal reverse specification strategy to the field of OO.

5.3 Overview

The overall approach can be described as follows



In 4.2 we have expressed that one of the most important weaknesses the presented formal reverse strategies have is that they are applied in a monolithic way on all the program. In this work, we try to solve this problem by using slicing techniques. As we have said in 2.2, a slice captures all the computations on a given variable. Thus, the sentences extracted really constitute a program that executes a subfunctionality of the overall program [1].

We have already expressed that, conceptually, reverse software specification is the converse of forward software reification [40]. Therefore, we would expect to define the formal semantics of a target language, applying it to the legacy code and following the process with formal reasoning. The approach can be applicable on toy source codes, not on real life source codes.

So, we propose the use of slicing techniques to reduce the complexity of handling the formal semantics of all program variables at the same time. In order to found our proposal, we develop the relation between denotational semantics and program synthesis afterward.

5.4 The RPC process

In 2.3.3 we have already indicated that denotational semantics captures the behavior of P, that is, $[\![P]\!]$ (recall (2.2)) by means of the input/otput relation. We assume the initial data on which the computer is to operate (the input) are placed in the initial state; the results of the computation (the output) are found subsequently, in the final state [30]. In this way, $[\![P]\!]$ expresses the change of state (*i.e.* the set of variables which P has access to) that occurs when P is executed.

[1]It is important to notice that when we mention variables in this context, we refer to output variables.

In formal forward engineering (refinement) begins from a specification succesively rewritten until a semantic S_n is found

$$S \supseteq S_1 \supseteq \ldots \supseteq S_n = [\![P]\!] \tag{5.1}$$

The constructions obtained can be handled by means of the available program combinators, *e.g.* either operator (3.16). So, for example, in

$$[\![if\ C\ then\ P\ else\ Q]\!] = [[\![P]\!], [\![Q]\!]] \cdot C?$$

$[,]$ denotes the alternative execution of P and Q depending on C. Thus, if the specification S can be rewritten as $[M, N] \cdot C?$ and we find that $M \supseteq [\![P]\!]$ if C is true, and $N \supseteq [\![Q]\!]$ if C is false then $S \supseteq [\![if\ C\ then\ P\ else\ Q]\!]$ is a valid refinement step.

On the contrary, in reverse engineering the *program analysis technique* should go the other way round: try and identify alternative segments of the code so that their semantics can be inferred and abstracted upon. However, there is a problem in taking this way: going in the oppossite direction to (5.1) one may introduce arbitrary nondeterminism and obtain an imprecise specification S.

As we know, the state of the program is constituted by variables {monads?}

$$(v_1, \ldots, v_n) :: V$$

where

$$V = V_1 \times \ldots \times V_n$$

and V_i are datatypes. Each variable is involved in semantic rules based on the state-vector transformations $f :: V \longleftarrow V$. The semantic rules for each variable can be identified and grouped in independent computations

$$f_i :: V_i \longleftarrow V \tag{5.2}$$

Therefore, the whole semantics can be expressed following (3.4)

$$f = \langle f_1, \ldots, f_n \rangle \tag{5.3}$$

In this way, each computation f_i is self-sufficient and smaller than the original source code, and hence easier to reverse calculate [40].

In algebra of programming, a function that generates a vector of n results can always be transformed into a "split" (3.4) of n mutually dependent functions. This "splitting" effect can be handled by the *mutual recursion law* (3.58).

But, up to what point can the program slicing technique help in the **RPC** process?. Can we reconstrut the whole program semantics following (5.3)?

> **Conjecture:** Let P be a program exhibiting n output variables, and let $P_1, \ldots, P_n$ be the corresponding slices. Then the semantics of P can be recovered by combining these and only these slices. Formally
>
> $$[\![P]\!] = \langle [\![P_1]\!], \ldots, [\![P_n]\!] \rangle \tag{5.4}$$
>
> Therefore, slicing is a semantically sound code-decomposition technique.

However, (5.4) is false in the presence of non-termination. Weiser's algorithm [56] is no guarantee that the calculated slice will fail to halt whenever the original program fails to halt. This means that a slice can terminate in a state where the original did not. In terms of the semantics, this means Weiser's algorithm does not preserve a projection of the program semantics.

5.4.1 Non terminate situations

We can represent the semantics of loops as

$$[\![while\ c\ do\ s]\!] \stackrel{def}{=} \lambda\omega\sigma. \begin{cases} \omega([\![s]\!]\sigma) & if\ [\![c]\!]\sigma \\ \sigma & otherwise \end{cases} \tag{5.5}$$

where ω is the *least fixed point* of the function. So, for example, if we have $[\![while\ true\ do\ x := 1]\!]$ we would have

$$F = \lambda\omega\sigma. \begin{cases} \omega([\![x := 1]\!]\sigma) & if\ [\![true]\!]\sigma \\ \sigma & otherwise \end{cases} \tag{5.6}$$

as we are interested in the *least fixed point*

$$\omega = \lambda\sigma. \begin{cases} \omega([\![x := 1]\!]\sigma) & if\ [\![true]\!]\sigma \\ \sigma & otherwise \end{cases} \tag{5.7}$$

because the function evaluation is strict (recall 2.3.2)

$$[\![true]\!]\sigma = \begin{cases} \perp & if\ \sigma = \perp \\ true & otherwise \end{cases} \tag{5.8}$$

Therefore, $\lambda\sigma. \perp$ is a fixed point of F and hence the least fixed one. Consequently, the program *while true x:=1* will fail to terminate, without mattering which is its initial state.

Let us suppose the next program P = *while true x:=1; y:=1.* As we know that the semantics of the sequence of statements is

$$[\![s_1; s_2]\!] = \lambda\sigma.[\![s_2]\!]([\![s_1]\!]\sigma) \tag{5.9}$$

so, in our example we have

$$[\![P]\!] = \lambda\sigma.[\![y := 1]\!]([\![while\ true\ do\ x := 1]\!]\sigma)$$

as we know that the $while - do$ fails to terminate and applying λ-transformations (recall 2.3.2) we have

$$[\![y := 1]\!] \perp \tag{5.10}$$

the semantics of an assigment statement is finally defined as

$$[\![x := e]\!] = \lambda\sigma.\sigma[x \leftarrow [\![e]\!]\sigma]$$

Thus, (5.10) is rewritten as

$$\perp [y \leftarrow [\![1]\!] \perp]$$

Therefore, the semantics of our example is $\perp$. Let us suppose now that we calculate the slice on y in the current example. This slice is only composed by a $y := 1$ sentence. Consequently, $[\![P]\!] \subseteq [\![y := 1]\!]$. This means that in (5.4) we should replace = with $\subseteq$ and move to *relation algebra* [40].

At the same time, we can observe that we can calculate two slices from the previous example. The second slice on x would be *while true x:=1*. Thus, in (5.4) we should have

$$[\![P]\!] = \langle [\![y := 1]\!], [\![while\ true\ x := 1]\!] \rangle$$

We have already demostrated that $[\![P]\!]$ is $\perp$. Since we also have demostrated that $[\![while\ true\ x := 1]\!]$ is $\perp$, all the right hand side is $\perp$. Therefore, we have $\perp$, an undefined relation, on both sides.

5.4.2 Accumulation parameter introduction

Usually, the semantics of a block B, where some statements like *while, for* or *do* repeatedly execute a set of statements, will be inductively defined over some input type T like *a list, an array, a tree*. This type should be removed from the space vector

$$[\![B]\!] :: V \longleftarrow T \times V$$

As we have already shown in (5.2), slicing techniques provide us a collection of slices B_i

$$[\![B_i]\!] :: V_i \longleftarrow T \times V_i \tag{5.11}$$

Our intention is to abstract into inductively defined functions with signature

$$B_i :: V_i \longleftarrow T \tag{5.12}$$

The transformation from (5.12) to (5.11) is called *accumulation parameter introduction*. The idea behind this technique is to improve the efficiency of functional programs. Since our aim is to calculate abstract formal specifications, we are interested in the reverse application of this law. Therefore, we want to remove the accumulations.

5.5 Conditioned program slicing

5.6 The role of algebra

Having paved the way by program slicing techniques, the algebra of programming will now be able to handle the semantics acquired for each one of them. The splitting power of the slicing technique is evident when

the "interleaved" source code is "unraveled".

Due to efficiency reasons, programmers usually combine in the same programming construct, a loop for example, two or more logically independent computations. So, two or more loop constructions that can be executed in sequencial order, are merged in the same loop structure. This way for programmers

$$loop(f \cdot g) \textit{ is better than } (loop\ f) \cdot (loop\ g) \tag{5.13}$$

Obviously for human understanding

$$(loop\ f) \cdot (loop\ g) \textit{ is better than } loop(f \cdot g)$$

This "trick" to gain efficiency is one kind of formal transformation techniques known as *fusion laws*. Specifically, (5.13) can be handled formally by (3.3) which we have already developed previously. At this point, it is important to be clear that *loop f* and *loop g* are functional code blocks, consequently, their semantics can be expressed by functions.

Another different situation happens when several variables are computed in the same loop . Such variables can be independent among them but they all share the same visit to the underlying inductive datatype. In other words, we are referring to independent (perhaps) computations performed within the same loop. So, the control threads are executed in parallel. This can be formally handled by (3.4) as follows

$$loop\langle f, g\rangle = \langle loop\ f, loop\ g\rangle \tag{5.14}$$

Here the splitting power of the slicing technique is revealed in full. As we shall show later on, we will find the *mutual recursion law* (3.58) related to this law.

5.7 Case Studies

1. *More examples (change language / domain)*

   JNO to find a monadic example

2. *Topics addressed in Braga recently:*

   (a) *Conditional splicing as kind of "co-slicing", cf eithers (McCarthy) dualizing splits (a **paper** here?)*

   (b) *Slicing versus encapsulation and modularization*

   (c) *Is I/O to be sliced? That is, is `stdout` to become explicit?*

The source code and its respective slices are listed in appendix A and appendix B respectively.

5.8 Case study I

In 2.3.3 we have indicated that denotational semantics is the most appropriate technique for the formal reverse engineering strategy we are proposing here. In the current case study the semantics will be expressed in **H**ASKELL as follows:

Slicing criterion: $< 28, sum >$ (recall section 2.2)

```
rec([],test0,possum,negsum)  = ([],test0,possum,negsum)
rec(n:l,test0,possum,negsum) =
 if n>0
 then rec(l,test0,possum+n,negsum)
 else if n<0
      then rec(l,test0,possum,negsum-n)
      else if test0
           then if possum>=negsum
                then rec(l,test0,0,negsum)
                else rec(l,test0,possum,0)
           else rec(l,test0,possum,negsum)
```

```
ps(l,test0) =
 if trd(rec(l,test0,0,0))>frh(rec(l,test0,0,0))
 then trd(rec(l,test0,0,0))
 else frh(rec(l,test0,0,0))
```

In order to introduce *mutual recursion* (3.58) some previous transformations are executed.

```
loop([],test0,pair)  = ([],test0,pair)
loop(n:l,test0,pair) =
 loop(l,test0,accum(n,test0,pair))


accum(n,test0,(possum,negsum)) =
 if n>0
 then (possum+n,negsum)
 else if n<0
      then (possum,negsum-n)
      else if test0
           then if possum>=negsum
                then (0,negsum)
                else (possum,0)
           else (possum,negsum)
```

Thus, we can remove the accumulator from *loop_trans*

```
loop_trans_s([],test0)  = (0,0)
loop_trans_s(n:l,test0) =
 accum(n,test0,loop_trans_s(l,test0))


ps_trans(l,test0) =
```

```
 if fst(loop_trans_s(reverse l,test0))>
    snd(loop_trans_s(reverse l,test0))
 then fst(loop_trans_s(reverse l,test0))
 else snd(loop_trans_s(reverse l,test0))
```

So, the *loop_trans_* function can be expressed as a catamorphism

$$(\![\langle \underline{0}, \underline{0} \rangle, accum]\!) \tag{5.15}$$

Re-structuring *accum* function

```
accum_ps(n,test0,(possum,negsum)) =
 if n>0
 then (possum+n,negsum)
 else reset_s(n,test0,(possum,negsum))


accum_ns(n,test0,(possum,negsum)) =
      if n<0
      then (possum,negsum-n)
      else reset_s(n,test0,(possum,negsum))


reset_s (n,test0,(possum,negsum)) =
  if n==0 && test0
  then if possum>=negsum
       then (0,negsum)
       else (possum,0)
  else (possum,negsum)
```

Following with the transformations on the last functions

```
accum_psa(n,test0,(ps,ns)) =
```

```
 if n>0
 then ps+n
 else reset_ps(n,test0,(ps,ns))

reset_ps(n,test0,(ps,ns)) =
 if n==0 && test0 && ps>ns
 then 0
 else ps

accum_nsa(n,test0,(ps,ns)) =
 if n<0
 then ps-n
 else reset_ns(n,test0,(ps,ns))

reset_ns(n,test0,(ps,ns)) =
 if n==0 && test0 && ns>ps
 then 0
 else ns
```

we reformulate (5.15)

$$(\![\langle \underline{0}, \underline{0}\rangle, \langle accum_psa, accum_nsa\rangle]\!) \tag{5.16}$$

then, by *exchange law* (3.25), we get

$$(\!\langle [\underline{0}, accum_psa], [\underline{0}, accum_nsa]\rangle\!) \tag{5.17}$$

In order to complete the specifications of $< 28, sum >$ slice

```
final_ifs(f,s) = if f>s
```

```
                 then f
                 else s


ps_trans2(l,test0) =
 (final_ifs . loop_trans_s) (reverse l,test0)
```

The application of *mutual recursion* (3.58) is only possible reversing the input list. So the final formal specification of $< 28, sum >$ slice is a composition of two functions (3.3)

$$final_ifs \cdot loop_trans \tag{5.18}$$

A similar process is applied on $< 29, prod >$ slice. Afterwards, the sematics of this slice is shown in HASKELL :

Slicing criterion: $< 29, prod >$

```
rec_p([],test0,posprod,negprod)  = ([],test0,posprod,negprod)
rec_p(n:l,test0,posprod,negprod) =
 if n>0
 then rec_p(l,test0,posprod*n,negprod)
 else if n<0
      then rec_p(l,test0,posprod,negprod*(-n))
      else if test0
           then if posprod>=negprod
                then rec_p(l,test0,1,negprod)
                else rec_p(l,test0,posprod,1)
           else rec_p(l,test0,posprod,negprod)


prod(l,test0) =
 if trd(rec_p(l,test0,1,1))>frh(rec_p(l,test0,1,1))
 then trd(rec_p(l,test0,1,1))
```

```
 else frh(rec_p(l,test0,1,1))
```

Some transformations are immediately executed

```
loop_p([],test0,pair)  = ([],test0,pair)
loop_p(n:l,test0,pair) =
 loop_p(l,test0,accum_p(n,test0,pair))


accum_p(n,test0,(posprod,negprod)) =
 if n>0
 then (posprod*n,negprod)
 else if n<0
      then (posprod,negprod*(-n))
      else if test0
           then if posprod>=negprod
                then (1,negprod)
                else (posprod,1)
           else (posprod,negprod)
```

Removing the accumulator from *loop_p*

```
loop_trans_p([],test0)  = (1,1)
loop_trans_p(n:l,test0) =
 accum_p(n,test0,loop_trans_p(l,test0))


prod_trans(l,test0) =
 if fst(loop_trans_p(reverse l,test0))>
    snd(loop_trans_p(reverse l,test0))
 then fst(loop_trans_p(reverse l,test0))
 else snd(loop_trans_p(reverse l,test0))
```

So, *loop_trans_p* can be expressed as a list catamorphism

$$(\![\langle \underline{1}, \underline{1} \rangle, accum_p]\!) \tag{5.19}$$

Finally, the *accum_p* function is re-structured as follows

```
accum_pp(n,test0,(posprod,negprod)) =
 if n>0
 then (posprod*n,negprod)
 else reset_p(n,test0,(posprod,negprod))


accum_np(n,test0,(posprod,negprod)) =
      if n<0
      then (posprod,negprod*(-n))
      else reset_p(n,test0,(posprod,negprod))


reset_p (n,test0,(posprod,negprod)) =
  if n==0 && test0
  then if posprod>=negprod
       then (1,negprod)
       else (posprod,1)
  else (posprod,negprod)
```

Following with the transformations on these functions

```
accum_ppa(n,test0,(pp,np)) =
 if n>0
 then pp*n
 else reset_pp(n,test0,(pp,np))
```

```
reset_pp(n,test0,(pp,np)) =
 if n==0 && test0 && pp>np
 then 1
 else pp

accum_npa(n,test0,(pp,np)) =
 if n<0
 then np*(-n)
 else reset_ns(n,test0,(pp,np))

reset_ns(n,test0,(pp,np)) =
 if n==0 && test0 && np>pp
 then 1
 else np
```

At this point (5.19) is re-written as

$$(\![\langle \underline{1}, \underline{1}\rangle, \langle accum_ppa, accum_npa\rangle]\!) \tag{5.20}$$

By *exchange law* (3.25)

$$(\!\langle [\underline{1}, accum_ppa], [\underline{1}, accum_npa]\rangle\!) \tag{5.21}$$

In order to complete the transformations

```
final_ifp(f,s) = if f>s
                 then f
                 else s

prod_trans_p(l,test0) =
 (final_ifp . loop_trans_p ) (reverse l,test0)
```

and finally, by (3.3), we get the $< 29, prod >$ slice specification

$$final_ifp \cdot loop_trans_p \tag{5.22}$$

Based on (5.44) and (5.48) we can construct the whole program specification following *split operator* (3.4)

$$\langle final_ifs \cdot loop_trans_s, final_ifp \cdot loop_trans_p\rangle \tag{5.23}$$

and by $\times$-*absorption property* (3.10) in reverse order

$$(final_ifs \times final_ifp) \cdot \langle loop_trans_s, loop_trans_p\rangle \tag{5.24}$$

It is important to notice that in this case the reverse calculation process can also be applied in a *transversal* way: *transversal reverse calculation process.* On one hand, we can observe that (5.17) and (5.20) have the same signature: $Int \times Int \longleftarrow Int^*$. So, they may be composed by (3.4)

$$\langle loop_trans_s, loop_trans_p\rangle \tag{5.25}$$

On the other hand, the two final conditionals are independent from each other. So, they can be computed in parallel. Therefore, *functional product* (3.8) is applicable

$$final_ifs \times final_ifp \tag{5.26}$$

Because the conditionals are executed after the iteration segment generates their results, (5.26) and (5.25) can be combined by means of (3.3). In this way, the same expression (5.24) comes out.

5.9 Case Study II: Monads.

```
--------------------
-- SLICE ON prevc --
--------------------
slc_prevc =
```

```
 readFile "input.txt" >>=
 (\s -> case s of
         []     -> print "empty file"
         (x:xs) -> l_prevc x xs >>= \r -> print r)

l_prevc prevc []     = return(prevc)
l_prevc prevc (x:xs) =
  if (prevc=='\r') && (x=='\n')  -- where '\r'=carriage return (chr 13)
  then if (check_end xs)         --       '\n'=line feed (chr 10)
       then return(prevc)    -- Break
       else l_prevc (head xs) (tail xs)
  else l_prevc x xs

check_end []     = True
check_end (x:xs) = False

---------------------
-- SLICE ON lns_dn --
---------------------
slc_lnsdn =
 readFile "input.txt" >>=
 (\s -> case s of
         []     -> print "empty file"
         (x:xs) -> l_lnsdn x xs 0 >>= \r -> print r)

l_lnsdn prevc [] lnsdn     = return(prevc,lnsdn)
l_lnsdn prevc (x:xs) lnsdn =
  if (prevc=='\r') && (x=='\n')
```

```
  then if (check_end xs)
       then return(prevc,lnsdn+1)    -- Break
       else l_lnsdn (head xs) (tail xs) (lnsdn+1)
  else l_lnsdn x xs (lnsdn)


---------------------
-- SLICE ON chs_rd --
---------------------
slc_chsrd =
 readFile "input.txt" >>=
 (\s -> case s of
         []     -> print "empty file"
         (x:xs) -> l_chsrd x xs 1 >>= \r -> print r)

l_chsrd prevc [] chsrd     = return(prevc,chsrd)
l_chsrd prevc (x:xs) chsrd =
  if (prevc=='\r') && (x=='\n')
  then if (check_end xs)
       then return(prevc,chsrd)    -- Break
       else l_chsrd (head xs) (tail xs) (chsrd+1)
  else l_chsrd x xs (chsrd+1)

---------------------
-- SLICE ON chs_pr --
---------------------
slc_chspr =
 readFile "input.txt" >>=
 (\s -> case s of
```

```
         []     -> print "empty file"
         (x:xs) -> l_chsrd x xs 1 >>=
                   \r -> (final_check (fst(r)) (snd(r)) >>=
                          \q -> print q))


l_chspr prevc [] chspr     = return(prevc,chspr)
l_chspr prevc (x:xs) chspr =
  if (prevc=='\r') && (x=='\n')
  then if (check_end xs)
       then return(chr 4,chspr+1)  -- (chr 4: End of Transmission)
       else l_chspr (head xs) (tail xs) (chspr+1)
  else l_chspr x xs (chspr+1)


final_check prevc chspr =
 if process_cntl_z == "ASISCZ"
 then if prevc == (chr 4)
      then return(chspr+1)
      else return(chspr)
 else if process_cntl_z == "CHKCZ"
      then if (prevc/=chr 4) && (prevc/=chr 032)
           then return(chspr+1)
           else return(chspr)
      else return(chspr)


---------------------
-- SLICE ON output --
---------------------
slc_output =
```

```
 readFile "input.txt" >>=
 (\s -> case s of
         []     -> print "empty file"
         (x:xs) -> openFile "output.txt" WriteMode >>=
                    \h -> l_output x xs h >>=
                          \h -> final_output (fst(h)) (snd(h)) >>=
                                \h1 -> do hClose h1
                                          openFile "output.txt" ReadMode >>=
                                             \h3 -> hGetContents h3 >>=
                                                    \x -> putStr x)

-- the last three lines can also be writes as follows
--                                           h3 <- openFile "output.txt" ReadMode
--                                           x <- hGetContents h3
--                                           putStr x)

l_output prevc [] h     = return(prevc,h)
l_output prevc (x:xs) h =
  if (prevc=='\r') && (x=='\n')
  then do hPutChar h '\n'
          if (check_end xs)
             then return(prevc,h)
             else l_output (head xs) (tail xs) h
  else do hPutChar h prevc
          l_output x xs h

final_output prevc h =
 if process_cntl_z == "ASISCZ"
```

```
 then if prevc /= (chr 4)

      then do hPutChar h prevc

              return(h)

      else return(h)

 else if process_cntl_z == "CHKCZ"

      then if (prevc/=chr 4) && (prevc/=chr 032)

           then do hPutChar h prevc

                   return(h)
```

5.10 Case study III: Conditional slicing in the reverse calculation process

So far, we have only considered *static slicing* as a mechanism to handle complexity. But the strategy that we are proposing can articulate other kinds of slicing techniques with *algebra of programming*. Specifically, in this section we develop an example where we use *conditioned slices* [5].

The source code used here has also been used in section (5.8), and its *conditioned slices* are listed in appendix (B.4).

The following **H**ASKELL code is the semantics of each slice calculated from each conditioned program. The slice on *possum* is considered first

```
----------------------------

-- First conditioned program

----------------------------

-- Slice on sum: calculate the sum of positive numbers.

accum_cs([]) = 0

accum_cs (n:l) =

 if n>0
```

```
 then n + accum_cs(l)
 else accum_cs(l)

final_cifs(possum) =
 if possum>0
 then possum
 else undefined

pc1cs1(l) =
 (final_cifs . accum_cs) (l)
```

From here, we calculate the corresponding formulae to *accum_cs*

$$(\![\underline{0}, (>0) \cdot \pi_1 \longrightarrow add(\pi_1) \cdot \pi_2, \pi_2]\!) \tag{5.27}$$

where we apply (3.2), (3.16), (3.29) and (3.40). The complete algebraic specification of this slice is obtained by (3.3)

$$final_cifs \cdot accum_cs \tag{5.28}$$

The next **HASKELL** code expresses the semantics of the slice on *posprod* in the first conditioned program

```
-- Slice on prod: calculate the product of positive numbers.
accum_cp([]) = 0
accum_cp (n:l) =
 if n>0
 then n * accum_cp(l)
 else accum_cp(l)

final_cifp(posprod) =
```

```
 if posprod>1
 then posprod
 else undefined


pc1cs2(l) =
 (final_cifp . accum_cp) (l)
```

As in the previous case, we calculate the formula for *accum_cp*

$$(\![\underline{1}, (>0) \cdot \pi_1 \longrightarrow mult(\pi_1) \cdot \pi_2, \pi_2]\!) \tag{5.29}$$

The same set of algebraic laws used in (5.27) is used. The complete specifications of this slice is as follows

$$final_cifp \cdot accum_cp \tag{5.30}$$

where, again, (3.3) is applied.

Now we begin to consider the second conditioned program. The semantics of the slice on *negsum* calculated from this program is as follows

```
-----------------------------
-- Second conditioned program
-----------------------------
-- Slice on sum: calculate the sum of negative numbers.
accum_cn([]) = 0
accum_cn (n:l) =
 if n<0
 then (-n) + accum_cn(l)
 else accum_cn(l)
```

```
final_cifn(negsum) =
 if 0>negsum
 then undefined
 else negsum


pc2cs1(l) =
 (final_cifn . accum_cn) (l)
```

The algebraic format for *accum_cn* is

$$(\![\underline{0}, (< 0) \cdot \pi_1 \longrightarrow add(-\pi_1) \cdot \pi_2, \pi_2]\!) \tag{5.31}$$

As before, the set of algebraic laws is the same. And the final specification is expressed as

$$final_cifn \cdot accum_cn \tag{5.32}$$

The second slice in the second conditioned program is calculated on the variable *negprod*

```
-- Slice on prod: calculate the product of negative numbers.
accum_cnp([]) = 0
accum_cnp (n:l) =
 if n<0
 then (-n) * accum_cnp(l)
 else accum_cnp(l)


final_cifnp(negprod) =
 if 1>negprod
 then undefined
 else negprod
```

```
pc2cs2(l) =
 (final_cifnp . accum_cnp) (l)
```

The formula for *accum_cnp* is as follows

$$(\![\underline{1}, (< 0) \cdot \pi_1 \longrightarrow mult(\pi_1) \cdot \pi_2, \pi_2]\!) \tag{5.33}$$

And the complete algebraic specification for this slice is

$$final_cifnp \cdot accum_cnp \tag{5.34}$$

Finally, we considered the third conditioned program. Here, we calculate two slices again. Slice on *sum*

```
----------------------------
-- Third conditioned program
----------------------------
-- Slice on sum: reset to 0 possum and negsum

reset0([],test0,possum,negsum) = (possum,negsum)
reset0(n:l,test0,possum, negsum) =
 if n==0 && test0
 then if possum>negsum
      then reset0(l,test0,0,negsum)
      else reset0(l,test0,possum,0)
 else reset0(l,test0,possum,negsum)

final_ifrs(possum,negsum) =
 if possum>negsum
 then possum
 else negsum
```

```
pc3cs1(l,test0) =
 (final_ifrs . reset0) (l,test0,0,0)
```

In order to obtain an algebraic expression for this slice some transformations are required

```
reset0t([],test0,(possum,negsum)) = ([],test0,(possum,negsum))
reset0t(n:l,test0,(possum,negsum)) =
 reset0t(l,test0,set_sum(n,test0,(possum,negsum)))
```

where

```
set_sum(n,test0,(ps,ns)) =
 if n==0 && test0
 then if ps>ns
      then (0,ns)
      else (ps,0)
 else (ps,ns)
```

at this point, we can remove the accumulator

```
reset0tt([],test0) = (0,0)
reset0tt(n:l,test0) =
 set_sum(n,test0,reset0tt(l,test0))


pc3cs1t(l,test0) =
 (final_ifrs .  reset0tt) (l,test0)
```

So, we are ready to introduce mutual recursion (3.58)

```
set_ps(n,test0,(ps,ns)) =
 if n==0 && test0 ps>ns
```

```
 then 0
 else ps

set_ns(n,test0,(ps,ns)) =
 if (n==0 && test0 && ps<ns)
 then 0
 else ns
```

Hence, we can specify the recursion segment(*reset0tt*) as follows

$$(\!|[\langle \underline{0}, \underline{0} \rangle, \langle set_ps, set_ns \rangle]|\!) \tag{5.35}$$

by exchange law

$$(\!|\langle [\underline{0}, set_ps], [\underline{0}, set_ns] \rangle|\!) \tag{5.36}$$

Having introduced mutual recursion, we can obtain the complete algebraic specification of this slice

$$final_ifrs \cdot reset0tt \tag{5.37}$$

The same process is applied on slice on *prod*

```
-- Slice on prod: reset to 1 posprod and negprod
reset1([],test0,posprod,negprod) = (posprod,negprod)
reset1(n:l,test0,posprod, negprod) =
 if n==0 && test0
 then if posprod>negprod
      then reset1(l,test0,1,negprod)
      else reset1(l,test0,posprod,1)
 else reset1(l,test0,posprod,negprod)
```

```
final_ifrp(posprod,negprod) =
 if posprod>negprod
 then posprod
 else negprod


pc3cs2(l,test0) =
 (final_ifrp . reset1) (l,test0,1,1)
```

As before, some transformations are performed

```
reset1t([],test0,(posprod,negprod)) = ([],test0,(posprod,negprod))
reset1t(n:l,test0,(posprod,negprod)) =
 reset0t(l,test0,set_prod(n,test0,(posprod,negprod)))
```

where

```
set_prod(n,test0,(pp,np)) =
 if n==0 && test0
 then if pp>np
      then (0,np)
      else (pp,0)
 else (pp,np)
```

and then, removing the accumulator

```
reset1tt([],test0) = (1,1)
reset1tt(n:l,test0) =
 set_prod(n,test0,reset1tt(l,test0))


pc3cs2t(l,test0) =
 (final_ifrp .  reset1tt) (l,test0)
```

in order to introduce mutual recursion (3.58)

```
set_pp(n,test0,(pp,np)) =
 if n==0 && test0 pp>np
 then 1
 else pp


set_np(n,test0,(pp,np)) =
 if (n==0 && test0 && pp<np)
 then 1
 else np
```

Therefore, the recursion segment of this slice is specified as

$$(\![\langle \underline{1}, \underline{1} \rangle, \langle set_pp, set_np \rangle]\!) \tag{5.38}$$

by mutual recursion law (3.58)

$$(\!\langle [\underline{1}, set_pp], [\underline{1}, set_np] \rangle\!) \tag{5.39}$$

And, the complete algebraic specification of this slice is as follows

$$final_ifrp \cdot reset1tt \tag{5.40}$$

In summary, we have calculated six conditioned slices from the three conditioned programs; two from each conditioned program. At this point, we notice that our original conjecture is not working anymore. Similarly to the conjecture, the slices are calculated based on the output variables, but dissimilarly, each slice is extracted from a conditioned program, where some restrictions are imposed on the input data.

At first, we might think in reconstructing the whole program specification calculating beforehand the formal specification for each conditioned program. From this point, and by means of some algebraic laws,

we would combine these expressions and calculate the complete program specification. Unfortunately, and due to the restrictions imposed, each conditioned program can only contain a subset of the computations executed on the output variables. Thus, the handling of the algebraic specifications of each conditioned program as a whole is not possible. Therefore, calculus of the final specification in this way is inviable.

On the other hand, and in agreement with that expressed two paragraphs before, a static slice may be composed by two or more conditioned slices depending on the conditions. Hence, a reasonable approach to solve this problem would be to reconstruct the specification of the static slices from their corresponding conditioned slices. Since a static slice is also a program, we may expect to calculate the specification via the original conjecture. But again, this is not so. Because the conditioned slice calculus depends on the restrictions on the input data, the split operator (3.4) in the conjecture does not supply the appropriate structure. Therefore, either operator (3.16) must be used to decide which branch to follow according to the condition imposed. Simultaneously, the conjecture has the same problem we have referred to in the previous paragraph. The computations on the same variable are spread in different slices.

In synthesis, the solution would be a combination of what we indicated previously with the transversal reverse calculus of the section 5.8. So, in order to recover the formal specification for each static slice, we apply transversal reverse calculus on the conditioned slices that compose it.

First, we try to construct the algebraic specification of the recursive segment on the basis of (5.27), (5.31) and (5.36)

$$(\![[reset0tt, \langle accum_cs, accum_cn\rangle]]\!) \tag{5.41}$$

replacing *reset0tt* with their definition

$$(\![[\langle[\underline{0}, set_ps], [\underline{0}, set_ns]\rangle, \langle accum_cs, accum_ns\rangle]]\!) \tag{5.42}$$

by exchange law

$$sum = (\!|\langle[[\underline{0}, set_ps], accum_cs], [[\underline{0}, set_ns], accum_ns]\rangle|\!) \tag{5.43}$$

And finally, applying (3.3)

$$final_ifrs \cdot sum \tag{5.44}$$

Thus, we have reconstructed the algebraic specification for the static slice on *sum*. The same process is also applied on (5.29), (5.33) and (5.39)

$$(\!|[reset1tt, \langle accum_cp, accum_cnp\rangle]|\!) \tag{5.45}$$

replacing *reset1tt*

$$(\!|[\langle[\underline{1}, set_pp], [\underline{1}, set_np]\rangle, \langle accum_cp, accum_cnp\rangle]|\!) \tag{5.46}$$

by exchange law

$$prod = (\!|\langle[[\underline{1}, set_pp], accum_cp], [[\underline{1}, set_np], accum_cnp]\rangle|\!) \tag{5.47}$$

And finally by (3.3)

$$final_ifrp \cdot prod \tag{5.48}$$

At this point, we have recovered the algebraic specifications of the static slices on *sum* and *prod* based on their respective conditioned slices. The whole program specification is obtained combining (5.44) and (5.48) by (3.4)

$$\langle final_ifrs \cdot sum, fina_ifrp \cdot prod\rangle \tag{5.49}$$

and by (3.10)

$$(final_ifrs \times final_ifrp) \cdot \langle sum, prod\rangle \tag{5.50}$$

As we can observe, this specification is the same as the (5.24) obtained in section 5.8. As we have seen in this section, an important effort of transformation of the semantics in **H**ASKELLcode is necessary to apply algebraic laws. On the other hand, the need for transformations diminishes with conditioned slices but the complexity of applying algebraic laws increases. This is explained by the different size of the slices. When the slice size increases (static slices) the algebraic laws cannot handle its semantics and transformations are necessary. When the slice size diminishes (conditioned slices) the semantics is simpler and the algebraic laws can be applied almost immediately.

5.11 Evaluation

Cf the approach with the use of wide-spectrum languages (VDM++) Discuss with JNO, cf with ”secret” work cf JAVA

JNO to provide example patterns able to be reproduced and subject to transformations

5.12 Summary

Chapter 6

Conclusions and Future Work

Conclusions . . . .
. . . .
. . . .
. . . .
. . . .
. . . .

Future work:

Thesis has already started the work leading to a formal specification of a slicer. A lot more to be done.
Operational semantics instead of denotational semantics: will that make a difference?
Pointers? are these a problem?

- Talk about the possibility of applying multiple kinds of slicing techniques: conditioned slicing, amorphous slices, etc.
- The proposed approach requires an automatic support.
- How to use the formal specifications obtained in a reengineering process.

Appendix A

Appendix A: Source Code Examples

A.1 Example I

```
1  main() {
2  int a, test0, n, i, posprod, negprod, possum,
   negsum, sum, prod;
3  scanf("%d", &test0); scanf("%d", &n);
4  i = posprod = negprod = 1;
5  possum = negsum = 0;
6  while (i <= n) {
7        scanf("%d", &a);
8        if (a > 0) {
9           possum += a;
10          posprod *= a; }
11       else if (a < 0) {
12               negsum -= a;
13               negprod *= (-a); }
14            else if (test0) {
15                    if (possum >= negsum)
16                       possum = 0;
```

```
17                      else negsum = 0;
18                      if (posprod >= negprod)
19                         posprod = 1;
20                      else negprod = 1; }
21         i++; }
22 if (possum > negsum)
23    sum = possum;
24 else sum = negsum;
25 if (posprod >= negprod)
26    prod = posprod;
27 else prod = negprod;
28 printf ("%d \n", sum);
29 printf ("%d \n", prod); }
```

A.2 Example II: Monads

```
/* input & output files */
1  FILE *input; /* input file */
2  FILE *output; /* output file */
/* flags indicating command line options */
3  int processing_direction = TOUNIX; /* which filter */
4  int process_cntl_z = ASISCZ; /* process ^Z or not */
5  int verbose = QUIET; /* print report or not */
/* character and line counts */
6  int chs_rd = 0; /* count chars read */
7  int chs_pr = 0; /* count chars written */
8  int lns_dn = 0; /* count lines */
/*********************************************************************/
/* prototypes */
```

```
9  void dos_to_unix( void); /* CR-LF to newline */
/*****************************************************************/
/* procedures */
/*---------------------------------------------------------------*/
/* dos_to_unix() copy CR-LF to newline */
11 void dos_to_unix() /* CR-LF to newline */
12 {
13        int c; /* character to be copied */
14        int prev_c;      /* look ahead character */
        /* start prev_c, c pipeline */
15        if( (prev_c = fgetc(input)) == EOF ) /* check eof */
16        {
17  fprintf( stderr, "Empty input file\n");
18         return; /* quit if no work */
19        }
20        chs_rd++; /* count chars */
       /* copy character by character */
21        while( (c = fgetc(input)) != EOF ) /* read to eof */
22        {
23         chs_rd++; /* count chars */
        /* check for a CR-LF sequence */
24         if( (prev_c == CR) && (c == LF) ) /* found CR-LF */
25                {
26                    fputc( NL, output);  /* write newline */
27                    chs_pr++; /* count chars */
28                    lns_dn++; /* count lines */
29                    if( (c = fgetc(input)) == EOF ) /* reload pipeline */
30                        {
```

```
31                       prev_c = c; /* set flag */
32                       break; /* quit at eof */
33                      }
34                      chs_rd++;
35              }
36              else /* any other */
37              {
38                 fputc( (char) prev_c, output); /* write char */
39                 chs_pr++; /* count chars */
40              }
41              prev_c = c; /* cycle pipeline */
42       }
       /* cntl-Z as-is */
43        if( process_cntl_z == ASISCZ ) /* write last character */
44        {
45           if( prev_c != EOF )
46                {
47           fputc( (char) prev_c, output); /* write it */
48                 chs_pr++; /* count it */
49               }
50        }
51        else if( process_cntl_z == CHKCZ ) /* guarantee no ^Z */
52        {
53         if( (prev_c != EOF) && (prev_c != CZ) ) /* char is not ^Z */
54                {
55                   fputc( (char) prev_c, output); /* write it */
56                  chs_pr++; /* count it */
57                }
```

```
58        }
59        return; /* done dos_to_unix() */
60 }
```

A.3 Example III: Pointers

```
1  #include <string.h>
2  #include "bacstd.h"

3  int _CfnTYPE strtrimcr(char *szStr, char *szSet)
4  {
5        int   i, j;                             /* Locale counters */

      /*-----------------------------------------------------*/

6        j = i = strlen(szStr) - 1;         /* Find length of string */

7        while (strrchr(szSet, szStr[ i ]) && (0 <= i))
8        {
    /* While string is terminated by one of the specified characters */
9              szStr[ i-- ] = '\0';    /*- Replace character with '\0' */
10       }

11       return(j - i);    /* Return the difference between old and new length */
12 }



13 int _CfnTYPE strtrimcl(char *szStr, char *szSet)
14 {
```

```
15      int   i = 0, j;

    /*---------------------------------------------------*/

16      j = strlen(szStr) - 1;                    /* Find length of string */

17      while (strrchr(szSet, szStr[ i ]) && (i <= j))
18      {
           /* While first character in string matches tag */

19            i++;                    /*- Count no of removed chars */
20      }

21      if (0 < i)                    /* IF there were matches */
22         strcpy(szStr, &szStr[ i ]); /*- shift string to the left */

23      return(i);                    /* Return no of matching chars */
24 }


25 int _CfnTYPE strtrimc(char *szStr, char *szSet)
26 {
27      int   iStatusFlag;

    /*---------------------------------------------------*/

28      iStatusFlag =  strtrimcl(szStr, szSet);
29      iStatusFlag += strtrimcr(szStr, szSet);
```

```
30       return(iStatusFlag);
31 }



32 main()
33 {
34       char  *strl = "xxyzxxxLeading x",
35                   *strr = "x Traling xxyzxxx",
36                   *str  = "xxzyxLead-&trailingxzxyx";
      /*--------------------------------*/

37       printf("\nBefore convertion:\n\t\"%s\"\n\t\"%s\"\n\t\"%s\"\n",
                  strl, strr, str);

38       strtrimcr(strr, "xyz");
39       strtrimcl(strl, "xyz");
40       strtrimc( str, "xyz");

41       printf("\nAfter convertion:\n\t\"%s\"\n\t\"%s\"\n\t\"%s\"\n",
                  strl, strr, str);

42       return(0);
43 }
```

Appendix B

Appendix B: Calculated Slices

B.1 Calculated slices from Example I

Static slicing criterion: (28,sum)

```
1  main() {
2  int a, test0, n, i, possum,
   negsum, sum;
3  scanf("%d", &test0); scanf("%d", &n);
4  i = 1;
5  possum = negsum = 0;
6  while (i <= n) {
7        scanf("%d", &a);
8        if (a > 0) {
9           possum += a;}
11       else if (a < 0) {
12               negsum -= a;}
14            else if (test0) {
15                    if (possum >= negsum)
16                       possum = 0;
17                    else negsum = 0;}
```

```
21       i++; }
22 if (possum > negsum)
23    sum = possum;
24 else sum = negsum;
28 printf ("%d \n", sum);}
```

Static slicing criterion: (29,prod)

```
1  main() {
2  int a, test0, n, i, posprod, negprod,
   prod;
3  scanf("%d", &test0); scanf("%d", &n);
4  i = posprod = negprod = 1;
6  while (i <= n) {
7        scanf("%d", &a);
8        if (a > 0) {
10          posprod *= a; }
11       else if (a < 0) {
13               negprod *= (-a); }
14            else if (test0) {
18                    if (posprod >= negprod)
19                       posprod = 1;
20                    else negprod = 1; }
21       i++; }
25 if (posprod >= negprod)
26    prod = posprod;
27 else prod = negprod;
29 printf ("%d \n", prod); }
```

B.2 Dos to Unix: Monads

```
-----------------------
--- SLICE ON prev_c ---
-----------------------
/* input & output files */
1  FILE *input; /* input file */
2  FILE *output; /* output file */
/* flags indicating command line options */
3  int processing_direction = TOUNIX; /* which filter */
4  int process_cntl_z = ASISCZ; /* process ^Z or not */
5  int verbose = QUIET; /* print report or not */
/* character and line counts */
6  int chs_rd = 0; /* count chars read */
7  int chs_pr = 0; /* count chars written */
8  int lns_dn = 0; /* count lines */
/*********************************************************************/
/* prototypes */
9  void dos_to_unix( void); /* CR-LF to newline */
/*********************************************************************/
/* procedures */
/*-------------------------------------------------------------------*/
/* dos_to_unix() copy CR-LF to newline */
11 void dos_to_unix() /* CR-LF to newline */
12 {
13         int c; /* character to be copied */
14         int prev_c;     /* look ahead character */
15         if( (prev_c = fgetc(input)) == EOF ) /* check eof */
```

```
16          {
17  fprintf( stderr, "Empty input file\n");
18           return; /* quit if no work */
19          }
21          while( (c = fgetc(input)) != EOF ) /* read to eof */
22          {
24           if( (prev_c == CR) && (c == LF) ) /* found CR-LF */
25                  {
29                      if( (c = fgetc(input)) == EOF ) /* reload pipeline */
30                          {
31                           prev_c = c; /* set flag */
32                           break; /* quit at eof */
33                          }
35                  }
36                  else /* any other */
37                  {
40                  }
41                  prev_c = c; /* cycle pipeline */
42         }
59          return; /* done dos_to_unix() */
60 }


------------------
--- SLICE ON c ---
------------------
/* input & output files */
1  FILE *input; /* input file */
2  FILE *output; /* output file */
```

```
/* flags indicating command line options */
3  int processing_direction = TOUNIX; /* which filter */
4  int process_cntl_z = ASISCZ; /* process ^Z or not */
5  int verbose = QUIET; /* print report or not */
/* character and line counts */
6  int chs_rd = 0; /* count chars read */
7  int chs_pr = 0; /* count chars written */
8  int lns_dn = 0; /* count lines */
/*************************************************************************/
/* prototypes */
9  void dos_to_unix( void); /* CR-LF to newline */
/*************************************************************************/
/* procedures */
/*-----------------------------------------------------------------------*/
/* dos_to_unix() copy CR-LF to newline */
11 void dos_to_unix() /* CR-LF to newline */
12 {
13        int c; /* character to be copied */
14        int prev_c;      /* look ahead character */
        /* start prev_c, c pipeline */
15        if( (prev_c = fgetc(input)) == EOF ) /* check eof */
16        {
17  fprintf( stderr, "Empty input file\n");
18         return; /* quit if no work */
19        }
21        while( (c = fgetc(input)) != EOF ) /* read to eof */
22        {
24         if( (prev_c == CR) && (c == LF) ) /* found CR-LF */
```

```
25                  {
29                      if( (c = fgetc(input)) == EOF ) /* reload pipeline */
30                          {
31                           prev_c = c; /* set flag */
32                           break; /* quit at eof */
33                          }
35                  }
36                  else /* any other */
37                  {
40                  }
41                  prev_c = c; /* cycle pipeline */
42          }
59          return; /* done dos_to_unix() */
60 }


-----------------------
--- SLICE ON chs_rd ---
-----------------------
/* input & output files */
1  FILE *input; /* input file */
2  FILE *output; /* output file */
/* flags indicating command line options */
3  int processing_direction = TOUNIX; /* which filter */
4  int process_cntl_z = ASISCZ; /* process ^Z or not */
5  int verbose = QUIET; /* print report or not */
/* character and line counts */
6  int chs_rd = 0; /* count chars read */
7  int chs_pr = 0; /* count chars written */
```

```
8  int lns_dn = 0; /* count lines */
/*****************************************************************/
/* prototypes */
9  void dos_to_unix( void); /* CR-LF to newline */
/*****************************************************************/
/* procedures */
/*---------------------------------------------------------------*/
/* dos_to_unix() copy CR-LF to newline */
11 void dos_to_unix() /* CR-LF to newline */
12 {
13          int c; /* character to be copied */
14          int prev_c;      /* look ahead character */
15          if( (prev_c = fgetc(input)) == EOF ) /* check eof */
16          {
17  fprintf( stderr, "Empty input file\n");
18           return; /* quit if no work */
19          }
20          chs_rd++; /* count chars */
21          while( (c = fgetc(input)) != EOF ) /* read to eof */
22          {
23           chs_rd++; /* count chars */
24           if( (prev_c == CR) && (c == LF) ) /* found CR-LF */
25                  {
29                      if( (c = fgetc(input)) == EOF ) /* reload pipeline */
30                          {
31                           prev_c = c; /* set flag */
32                           break; /* quit at eof */
33                          }
```

```
34                          chs_rd++;
35                  }
36                  else /* any other */
37                  {
40                  }
41                  prev_c = c; /* cycle pipeline */
42         }
59          return; /* done dos_to_unix() */
60 }


-----------------------
--- SLICE ON chs_pr ---
-----------------------
/* input & output files */
1  FILE *input; /* input file */
2  FILE *output; /* output file */
/* flags indicating command line options */
3  int processing_direction = TOUNIX; /* which filter */
4  int process_cntl_z = ASISCZ; /* process ^Z or not */
5  int verbose = QUIET; /* print report or not */
/* character and line counts */
6  int chs_rd = 0; /* count chars read */
7  int chs_pr = 0; /* count chars written */
8  int lns_dn = 0; /* count lines */
/*********************************************************************/
/* prototypes */
9  void dos_to_unix( void); /* CR-LF to newline */
/*********************************************************************/
```

```
/* procedures */
/*----------------------------------------------------------------------*/
/* dos_to_unix() copy CR-LF to newline */
11 void dos_to_unix() /* CR-LF to newline */
12 {
13         int c; /* character to be copied */
14         int prev_c;      /* look ahead character */
         /* start prev_c, c pipeline */
15         if( (prev_c = fgetc(input)) == EOF ) /* check eof */
16         {
17  fprintf( stderr, "Empty input file\n");
18          return; /* quit if no work */
19         }
21         while( (c = fgetc(input)) != EOF ) /* read to eof */
22         {
24          if( (prev_c == CR) && (c == LF) ) /* found CR-LF */
25                 {
27                     chs_pr++; /* count chars */
29                     if( (c = fgetc(input)) == EOF ) /* reload pipeline */
30                         {
31                          prev_c = c; /* set flag */
32                          break; /* quit at eof */
33                         }
35                 }
36                 else /* any other */
37                 {
39                    chs_pr++; /* count chars */
40                 }
```

```
41                 prev_c = c; /* cycle pipeline */
42        }
43         if( process_cntl_z == ASISCZ ) /* write last character */
44         {
45            if( prev_c != EOF )
46                 {
48                  chs_pr++; /* count it */
49                }
50         }
51         else if( process_cntl_z == CHKCZ ) /* guarantee no ^Z */
52         {
53          if( (prev_c != EOF) && (prev_c != CZ) ) /* char is not ^Z */
54                 {
56                   chs_pr++; /* count it */
57                 }
58         }
59         return; /* done dos_to_unix() */
60 }


-----------------------
--- SLICE ON lns_dn ---
-----------------------
/* input & output files */
1  FILE *input; /* input file */
2  FILE *output; /* output file */
/* flags indicating command line options */
3  int processing_direction = TOUNIX; /* which filter */
4  int process_cntl_z = ASISCZ; /* process ^Z or not */
```

```
5  int verbose = QUIET; /* print report or not */
/* character and line counts */
6  int chs_rd = 0; /* count chars read */
7  int chs_pr = 0; /* count chars written */
8  int lns_dn = 0; /* count lines */
/*********************************************************************/
/* prototypes */
9  void dos_to_unix( void); /* CR-LF to newline */
/*********************************************************************/
/* procedures */
/*-------------------------------------------------------------------*/
/* dos_to_unix() copy CR-LF to newline */
11 void dos_to_unix() /* CR-LF to newline */
12 {
13         int c; /* character to be copied */
14         int prev_c;      /* look ahead character */
        /* start prev_c, c pipeline */
15         if( (prev_c = fgetc(input)) == EOF ) /* check eof */
16         {
17  fprintf( stderr, "Empty input file\n");
18          return; /* quit if no work */
19         }
21         while( (c = fgetc(input)) != EOF ) /* read to eof */
22         {
24          if( (prev_c == CR) && (c == LF) ) /* found CR-LF */
25                 {
28                     lns_dn++; /* count lines */
29                     if( (c = fgetc(input)) == EOF ) /* reload pipeline */
```

```
30                          {
31                           prev_c = c; /* set flag */
32                           break; /* quit at eof */
33                          }
35                  }
36                  else /* any other */
37                  {
40                  }
41                  prev_c = c; /* cycle pipeline */
42         }
59          return; /* done dos_to_unix() */
60 }
```

```
-----------------------
--- SLICE ON output ---
-----------------------
/* input & output files */
1  FILE *input; /* input file */
2  FILE *output; /* output file */
/* flags indicating command line options */
3  int processing_direction = TOUNIX; /* which filter */
4  int process_cntl_z = ASISCZ; /* process ^Z or not */
5  int verbose = QUIET; /* print report or not */
/* character and line counts */
6  int chs_rd = 0; /* count chars read */
7  int chs_pr = 0; /* count chars written */
8  int lns_dn = 0; /* count lines */
/*************************************************************************/
```

```
/* prototypes */
9  void dos_to_unix( void); /* CR-LF to newline */
/*******************************************************************/
/* procedures */
/*-----------------------------------------------------------------*/
/* dos_to_unix() copy CR-LF to newline */
11 void dos_to_unix() /* CR-LF to newline */
12 {
13        int c; /* character to be copied */
14        int prev_c;     /* look ahead character */
       /* start prev_c, c pipeline */
15        if( (prev_c = fgetc(input)) == EOF ) /* check eof */
16        {
17  fprintf( stderr, "Empty input file\n");
18         return; /* quit if no work */
19        }
21        while( (c = fgetc(input)) != EOF ) /* read to eof */
22        {
24         if( (prev_c == CR) && (c == LF) ) /* found CR-LF */
25                {
26                    fputc( NL, output);  /* write newline */
29                    if( (c = fgetc(input)) == EOF ) /* reload pipeline */
30                        {
31                         prev_c = c; /* set flag */
32                         break; /* quit at eof */
33                        }
35                }
36                else /* any other */
```

```
37                {
38                   fputc( (char) prev_c, output); /* write char */
40                }
41                prev_c = c; /* cycle pipeline */
42       }
        /* cntl-Z as-is */
43         if( process_cntl_z == ASISCZ ) /* write last character */
44         {
45            if( prev_c != EOF )
46                {
47            fputc( (char) prev_c, output); /* write it */
49               }
50         }
51         else if( process_cntl_z == CHKCZ ) /* guarantee no ^Z */
52         {
53          if( (prev_c != EOF) && (prev_c != CZ) ) /* char is not ^Z */
54                {
55                   fputc( (char) prev_c, output); /* write it */
57                }
58         }
59         return; /* done dos_to_unix() */
60 }
```

B.3 Removing string.(Pointers)

```
---------------------
--- SLICE ON strr ---
---------------------
```

```
1  #include <string.h>
2  #include "bacstd.h"


3  int _CfnTYPE strtrimcr(char *szStr, char *szSet)
4  {
5        int   i, j;
6        j = i = strlen(szStr) - 1;         /* Find length of string */
7        while (strrchr(szSet, szStr[ i ]) && (0 <= i))
8        {
9              szStr[ i-- ] = '\0';
10       }
11       return(j - i);
12 }

32 main()
33 {
34       char  *strr = "x Traling xxyzxxx";
37       printf("\nBefore convertion:\n\t\"%s\"\n\t\"%s\"\n\t\"%s\"\n",
                  strl, strr, str);
38       strtrimcr(strr, "xyz");
41       printf("\nAfter convertion:\n\t\"%s\"\n\t\"%s\"\n\t\"%s\"\n",
                  strr);

42       return(0);
43 }
```

```
--- SLICE ON strl ---
---------------------
1  #include <string.h>
2  #include "bacstd.h"

13 int _CfnTYPE strtrimcl(char *szStr, char *szSet)
14 {
15       int   i = 0, j;

16       j = strlen(szStr) - 1;
17       while (strrchr(szSet, szStr[ i ]) && (i <= j))
18       {
            /* While first character in string matches tag */
19          i++;                    /*- Count no of removed chars */
20       }
21       if (0 < i)                 /* IF there were matches */
22          strcpy(szStr, &szStr[ i ]); /*- shift string to the left */
23       return(i);                 /* Return no of matching chars */
24 }

32 main()
33 {
34       char  *strl = "xxyzxxxLeading x";
37       printf("\nBefore convertion:\n\t\"%s\"\n\t\"%s\"\n\t\"%s\"\n",
                  strl, strr, str);
39       strtrimcl(strl, "xyz");
41       printf("\nAfter convertion:\n\t\"%s\"\n\t\"%s\"\n\t\"%s\"\n",
                  strl);
```

```
42        return(0);
43 }
```

```
--------------------
--- SLICE ON str ---
--------------------
```

```
1  #include <string.h>
2  #include "bacstd.h"

3  int _CfnTYPE strtrimcr(char *szStr, char *szSet)
4  {
5         int   i, j;                             /* Locale counters */

6         j = i = strlen(szStr) - 1;         /* Find length of string */
7         while (strrchr(szSet, szStr[ i ]) && (0 <= i))
8         {
    /* While string is terminated by one of the specified characters */
9               szStr[ i-- ] = '\0';    /*- Replace character with '\0' */
10        }
11        return(j - i);    /* Return the difference between old and new length */
12 }



13 int _CfnTYPE strtrimcl(char *szStr, char *szSet)
14 {
15        int   i = 0, j;
```

```
16       j = strlen(szStr) - 1;          /* Find length of string */

17       while (strrchr(szSet, szStr[ i ]) && (i <= j))
18       {
19             i++;
20       }
21       if (0 < i)
22          strcpy(szStr, &szStr[ i ]);
23       return(i);             /* Return No of matching chars */
24 }


25 int _CfnTYPE strtrimc(char *szStr, char *szSet)
26 {
27       int   iStatusFlag;

28       iStatusFlag =  strtrimcl(szStr, szSet);
29       iStatusFlag += strtrimcr(szStr, szSet);
30       return(iStatusFlag);
31 }


32 main()
33 {
34       char  *str  = "xxzyxLead-&trailingxzxyx";
37       printf("\nBefore convertion:\n\t\"%s\"\n\t\"%s\"\n\t\"%s\"\n",
                  str);
40       strtrimc( str, "xyz");
```

```
41       printf("\nAfter convertion:\n\t\"%s\"\n\t\"%s\"\n\t\"%s\"\n",
                  str);
42       return(0);
43 }
```

B.4 Conditioned slicing

The following *conditioned slices* are calculated in (A.1) from the source code. First, the *conditioned programs*(recall section 2.2) are listed and then *conditioned slices* are calculated from them.

First conditioned program: The input condition applied is

$$\forall i, 1 \leq i \leq n, a_i > 0$$

```
1  main(){
2    int a, n, i, posprod, negprod, possum, negsum, sum, prod;
3    scanf("%d", &test0); scanf("%d", &n);
4    i = posprod = negprod = 1;
5    possum = negsum = 0;
6    while (i<=n) {
7        scanf("%d", &a);
8        if (a>0) {
9           possum += a;
10          posprod * = a; }
21       i++;}
22   if (posum >= negsum)
23      sum = possum;
25   if (posprod >= negprod)
26      prod = posprod;
28   printf("%d \n", sum);
29   printf("%d \n", prod);}
```

From here, we calculate the static slice on *sum*. Therefore, the slicing criterion would be

$$C = ((\forall i, 1 \leq i \leq n, a_i > 0), 28, \{sum\})$$

recall (2.1)

```
1  main(){
2    int a, n, i, possum, negsum, sum;
3    scanf("%d", &n);
4    i = 1;
5    possum = negsum = 0;
6    while (i<=n) {
7        scanf("%d", &a);
8        if (a>0) {
9           possum += a;}
21       i++;}
22   if (possum >= negsum)
23      sum = possum;
28   printf("%d \n", sum);}
```

We use the following criterion for the second static slice

$$C = ((\forall i, 1 \leq i \leq n, a_i > 0), 29, \{prod\})$$

```
1  main(){
2    int a, n, i, posprod, negprod, prod;
3    scanf("%d", &n);
4    i = posprod = negprod = 1;
6    while (i<=n) {
7        scanf("%d", &a);
8        if (a>0) {
```

```
10          posprod * = a; }
21       i++;}
25   if (posprod >= negprod)
26      prod = posprod;
29   printf("%d \n", prod);}
```

Second conditioned program: The input condition applied is

$$\forall i, 1 \leq i \leq n, a_i < 0$$

```
1  main(){
2    int a, test0, n, i, posprod, negprod, possum, negsum, sum, prod;
3    scanf("%d", &test0); scanf("%d", &n);
4    i = posprod = negprod = 1;
5    possum = negsum = 0;
6    while (i<=n) {
7        scanf("%d", &a);
8        if (a>0) {}
11       else if (a<0) {
12               negsum -= a;
13               negprod *= (-a);}
21       i++;}
22   if (possum >= negsum)
24   else sum = negsum;
25   if (posprod >= negprod)
27   else prod = negprod;
28   printf("%d \n", sum);
29   printf("%d \n", prod);}
```

We calculate a static slice on the second conditioned program following the next criterion:

$$C = ((\forall i, 1 \leq i \leq n, a_i < 0), 28, \{sum\})$$

```
1  main(){
2    int a, n, i, possum, negsum, sum;
3    scanf("%d", &n);
4    i = 1;
5    possum = negsum = 0;
6    while (i<=n) {
7        scanf("%d", &a);
8        if (a>0) {}
11       else if (a<0) {
12               negsum -= a;}
21       i++;}
22   if (possum >= negsum)
24   else sum = negsum;
28   printf("%d \n", sum);}
```

The second conditioned program is calculated based on

$$C = ((\forall i, 1 \leq i \leq n, a_i < 0), 29, \{prod\})$$

```
1  main(){
2    int a, n, i, posprod, negprod, prod;
3    scanf("%d", &n);
4    i = posprod = negprod = 1;
6    while (i<=n) {
7        scanf("%d", &a);
8        if (a>0) {}
11       else if (a<0) {
```

```
13              negprod *= (-a);}
21      i++;}
25  if (posprod >= negprod)
27  else prod = negprod;
29  printf("%d \n", prod);}
```

Third conditioned program : The input condition applied is

$$\forall i, i \leq i \leq n, a_i = 0 \wedge test0 = 1$$

```
1  main(){
2   int a, test0, n, i, posprod, negprod, possum, negsum, sum, prod;
3   scanf("%d", &test0); scanf("%d", &n);
4   i = posprod = negprod = 1;
5   possum = negsum = 0;
6   while (i<=n) {
7       scanf("%d", &a);
8       if (a>0) {}
11      else if (a<0) {}
14           else if (test0) {
15                   if (possum >= negsum)
16                      possum = 0;
17                   else negsum = 0;
18                   if (posprod >= negprod)
19                      posprod = 1;
20                   else negprod = 1;}
21      i++;}
22  if (posum >= negsum)
23     sum = possum;
24  else sum = negsum;
```

```
25   if (posprod >= negprod)
26      prod = posprod;
27   else prod = negprod;
28   printf("%d \n", sum);
29   printf("%d \n", prod);}
```

From this point, we calculate the next conditioned slice

$$C = ((\forall i, 1 \leq i \leq n, a_i = 0 \wedge test0 = 1), 28, \{sum\})$$

```
1  main(){
2    int a, test0, n, i, possum, negsum, sum;
3    scanf("%d", &test0); scanf("%d", &n);
4    i = 1;
5    possum = negsum = 0;
6    while (i<=n) {
7        scanf("%d", &a);
8        if (a>0) {}
11       else if (a<0) {}
14            else if (test0) {
15                    if (possum >= negsum)
16                       possum = 0;
17                    else negsum = 0;}
21       i++;}
22   if (possum >= negsum)
23      sum = possum;
24   else sum = negsum;
28   printf("%d \n", sum);}
```

The last conditioned slice is calculated based on

$$C = ((\forall i, 1 \leq i \leq n, a_i = 0 \land test0 = 1), 29, \{prod\})$$

```
1  main(){
2    int a, test0, n, i, posprod, negprod, prod;
3    scanf("%d", &test0); scanf("%d", &n);
4    i = posprod = negprod = 1;
6    while (i<=n) {
7        scanf("%d", &a);
8        if (a>0) {}
11       else if (a<0) {}
14            else if (test0) {
18                    if (posprod >= negprod)
19                       posprod = 1;
20                    else negprod = 1;}
21       i++;}
25   if (posprod >= negprod)
26      prod = posprod;
27   else prod = negprod;
29   printf("%d \n", prod);}
```

Bibliography

[1] S. Agerholm and P. G. Larsen. A lightweight to formal methods. In *International Worshop on Current Trends in Applied Formal Methods*, Springer-Verlag, Boppard, Germany, October 1998.

[2] H. Barendregt and E. Barendsen. Introduction to lambda calculus. *Workshop on Implementation of Functional Programming*, October 1994.

[3] R. Bird and O. de Moor. *Algebra of Programming.* Series in Computer Science. Prentice-Hall International, 1997. C. A. R. Hoare, series editor.

[4] Dines Bjorner. Formal methods in the 21'st century an assessment of today - predictions for the future, 1998.

[5] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information and Software Technology (special issue on program slicing)*, 40(11/12):595–607, 1998.

[6] B.H.C. Cheng and G.C. Gannod. A two-phase approach to reverse engineering using formal methods. In *Formal Methods in Programming and Their Applications.* Springer-Verlag, 1993. Lecture Notes in Computer Science.

[7] E.J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[8] Aniello Cimitile, Andrea De Lucia, and Malcolm C. Munro. A specification driven slicing process for identifying reusable functions. *Journal of Software Maintenance: Research and Practice*, 8(3):145–178, 1996.

[9] Edmund M. Clarke and Jeannette Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[10] Maarten M. Fokkinga. A gentle introduction to category theory: The calculational approach, June 1994.

[11] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[12] G. Gannod and B. H. C. Cheng. Abstraction of formal specifications from program code. In *Proceedings of the 3rd IEEE International Conference on Tools for Artificial Intelligence*, November 1991.

[13] G. C. Gannod and B. H. C. Cheng. A specification matching based approach to reverse engineering. In *21st International Conference on Software Engineering*, Los Angeles, CA, USA, May 1999.

[14] G.C. Gannod and B.H.C. Cheng. Strongest postcondition semantics as the formal basis for reverse engineering. In *1995 Work. Conference on Reverse Engineering*, pages 188–197, July 1995.

[15] G.C. Gannod and B.H.C. Cheng. A formal apprach for reverse engineering: a case study. In *WCRE'99*, 1999.

[16] Jeremy Gibbons. An introduction to the bird-meertens formalism. New Zeland Formal Program Development Colloquium Seminar, November 1994.

[17] Jeremy Gibbons. Conditionals in distributive categories. Technical report, Oxford Brooks University, August 1996.

[18] Jeremy Gibbons. Lecture notes on algebraic and co-algebraic methods for calculating functional programs, March 1999.

[19] Mike Gordon. Introduction to functional programming. Computing Laboratory, University of Cambridge, January 1996.

[20] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.

[21] M. Harman and S. Danicic. Amorphous program slicing. In *Proceedings of the 5th IEEE Internation Workshop on Program Comprehesion (IWPC'97) (Dearborn, Michigan, USA, May 1997)*, pages 70–79. IEEE CS Press, Los Alamitos, California, USA, 1997.

[22] M. Harman and K. B. Gallagher. Program slicing, 1998.

[23] Mark Harman, Sebastian Danicic, and Yoga Sivagurunathan. Program comprehension assisted by slicing and transformation, July 1995. In Malcolm Munro, editor, 1st Durham Workshop on Program Comprehension, Durham University, UK.

[24] E. De Hoffman, J. Charette, V. Stroobant, and J. Trottier. *Mass spectrometry: principles and applications.* John Wiley & Son Ltd., December 1996.

[25] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[26] IFAD. *The IFAD VDM-SL Language.* IFAD, Denmark, 1999.

[27] B. Korel and J. Laski. Dinamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[28] A. De Lucia, A. R. Fasolino, and M. Munro. Understanding functions behaviors through program slicing. In *Proceedings of 4th Workshop on Program Comprehension*, pages 9–18, Berlin, Germany, 1996. IEEE CS Press.

[29] C. Morgan. *Programming from Specification.* Series in Computer Science. Prentice-Hall International, 1990. C. A. R. Hoare, series editor.

[30] Carroll Morgan. *Programming from specification.* Prentice-Hall International, second edition, October 1998.

[31] Peter D. Mosses. Fundamental concepts and formal semantics of programming languages. BRICS and Department of Computer Science, University of Aarhus, Denmark, January 2002.

[32] F. L. Neves and J. N. Oliveira. Software Reuse by Model Reification . In *WISR'95 - 6th Annual Workshop on Software Reuse*, August 28–30 1995. Charles Il, Illinois, USA.

[33] F. L. Neves, J. C. Silva, and J. N. Oliveira. Converting informal meta-data to VDM-SL: A reverse calculation approach. In *VDM in Practice A Workshop co-located with FM'99: The World Congress on Formal Methods, Toulouse, France, 20-21 September*, September 1999.

[34] J. N. Oliveira. A reification calculus for model-oriented software specification. *Formal aspects on computing*, 2(1):1–23, April 1990.

[35] J. N. Oliveira. *Especificação Formal de Programas.* University of Minho, 1st edition, 1991. Lecture Notes for M.Sc. Course in Computing (in Portuguese; incl. extended abstract in English; last edition 1994).

[36] J. N. Oliveira. Software reification using the sets calculus. In *Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development, London, UK*, pages 140–171. Springer-Verlag, 8–10 January 1992. (Invited paper).

[37] J. N. Oliveira. A calculational approach to reverse specification, 1997. Seminar presented at UNU/IIST, Macau, May 13th, 1997, 22 p.

[38] J. N. Oliveira. A data structuring calculus and its application to program development, May 1998. Lecture Notes of M.Sc. Course Maestria em Ingeneria del Software, Departamento de Informatica, Facultad de Ciencias Fisico-Matematicas y Naturales, Universidad de San Luis, Argentina.

[39] J. N. Oliveira. *"Bagatelle in C arranged for VDM SoLo"* . In *Formal Aspects of Software Engineering (Colloquium to Mark the Retirement of Prof. Peter Lucas), Institute for Software Technology, Graz University of Technology*, 18-19th May 2001. To appear in the *Journal of Universal Computer Science.*

[40] J. N. Oliveira and G. Villavicencio. Reverse program calculation supported by code slicing. In *8th Working Conference on Reverse Engineering*, pages 35–45, Stuttgart, Germany, October 2001. IEEE CS Press.

[41] J.N. Oliveira. An introduction to pointfree programming, 1999. Departamento de Informtica, Universidade do Minho. 37p., chapter of book in preparation.

[42] Lawrence Paulson. Foundations of functional programming. Computing Laboratory, University of Cambridge.

[43] Benajmin C. Pierce. Foundational calculi for programming languages. *CRC handbook of Computer Science and Engineering*, December 1995.

[44] Dave Schmidt. Programming languages semantics. *ACM Computing Surveys*, 28(1):265–267, October 1996.

[45] C. F. Simpson and E. D. Katz. *Tandem Techniques.* John Wiley, Separation Science Series, 2000.

[46] M. P. Ward. Abstracting a specification from code. *Journal of Software Maintenance: Research and Practice*, 5:101–122, 1993.

[47] M. P. Ward. Transforming a program into specifications. Technical Report 88-1, Durham University, Jan. 1993.

[48] M. P. Ward. Reverse engineering through program transformation. *The Computer Journal*, 37(9):795–813, 1994.

[49] M. P. Ward. Specifications from source code:alchemists' dream or practical reality. In *4th Reengineering forum*, Victoria, Canada, September 1994. Reengineering forum.

[50] M. P. Ward. Program analysis by formal transformation. *The Computer Journal*, 39(7), 1996.

[51] M. P. Ward. Reverse engineering from assembler to formal specification via program transformation. In *7th Working Conference on Reverse Engineering*, Brisbane, Queensland, Australia, November 2000. IEEE CS Press.

[52] M. P. Ward. The transformational approach to source code analysis and manipulation. In *First International Workshop on Source Code Analysis and Manipulation*, Florence, Italy, November 2001. IEEE CS Press.

[53] M. P. Ward and K. H. Bennett. A practical program transformation system for reverse engineering. In *Working Conference on Reverse Engineering*, Baltimore, USA, May 1993. WCRE.

[54] M. P. Ward and K. H. Bennett. Formal methods for legacy systems. *Journal of Software Maintenance: Reseach and Practice*, 7(6):203–219, May-June 1995.

[55] Martin Ward. *Proving Program Refinement and Transformation*. PhD thesis, Oxford University, 1987.

[56] Mark Weiser. Program slicing. In *Fifth International Conference on Software Engineering, San Diego, California*, March 1981.