Reverse Program Calculation Supported by Code Slicing

G. Villavicencio

Universidad Católica de Santiago del Estero, Av. Alsina y Dalmacio Velez Sarsfield, 4200, Santiago del Estero, Argentina (gustavov@ucse.edu.ar)

J.N. Oliveira

Dep. Informática, Campus de Gualtar, Universidade do Minho 4700-320 Braga, Portugal (jno@di.uminho.pt)

Abstract

This paper sketches a discipline for reverse engineering which combines formal and semi-formal methods. Among the former is the algebra of programming*, which we apply in "reverse order" so as to reconstruct the formal specifications of legacy code. The latter includes* code slicing*, used as a means of trimming down the complexity of handling the formal semantics of all program variables at the same time.*

A strong point of the approach is its constructive style. Reverse calculations go as far as imploding auxiliary variables, introducing mutual recursion (if applicable) and transforming semantic functions into well-known generic programming schemata: cata/paramorphisms.

We illustrate this approach by reversing a piece of code (from C to HASKELL*) already studied in the code-slicing literature: the* WORDCOUNT *program.*

1 Introduction

This paper describes work in progress on the development of a discipline for reverse engineering which combines formal and semi-formal methods. The formal basis of the approach is the *algebra of programming* [3] which we intend to apply in "reverse order" so as to reconstruct the formal specifications of legacy code. Because of the complexity of reversing formal semantic descriptions of algorithmic code, we combine reverse algebraic calculation with code slicing [26] techniques.

This work is a follow up of the KARMA project [14], which has addressed *data reverse engineering* (DRE) in a similar way. The KARMA tool [14] synthesizes not only the specification but also the overall abstraction function of legacy data implicit in its reverse calculation.

2 Reification versus Reverse Specification

Software *reification* (refinement) is the process of converting abstract software specifications into real applications intended to run on some target hardware. The more colloquial term *refinement* is often preferred to "reification", after "reify", the process of something becoming real [1]. Such a process is synthetic and one-to-many in the sense that the same specification can lead to many reifications (implementations) depending on a myriad of design decisions which include choice of programming paradigm, selection of a specific language and so on. Reification is therefore a form of *forward engineering*, understood as the development of a system *by moving from abstract specifications to detailed implementations* [4].

Reverse specification is the opposite of reification. It is the analytical process of inferring the original specification (which actually may have never been written) of some running piece of software. It therefore is a a form of *reverse engineering*, understood as the *analysis of a system in order to identify components and intended behaviour in order to create higher level abstractions of the system* [4].

Software refinement is carried out with a variable concern for formality. If performed on a sound math-

[1] These two terms will be used interchangeably in this paper.

ematical basis, it becomes perhaps the most relevant branch of the formal method which supports it. However, *formal refinement* is not widely used due to the difficulty of scaling up its mathematical reasoning to real size problems. By contrast, informal refinement enforces so little discipline in software design that results are often catastrophic, particularly in the case of very large applications which have a long life-span. As a rule, such applications lack in documentation, have bugs and raise serious maintenance problems. Using or selling them becomes risky business.

Such bad *forward software engineering* standards sooner or later call for *reverse software engineering*, understood as the process of constructing descriptions at a higher level of abstraction of the components of a (large) software system which is in a "legacy" or "geriatric" state, thus facilitating the understanding of the system [8]. One thus gets into "geriatric" computing, or "infocare", a fast growing sector of information technology which, according to the terminology of [4], encompasses not only reverse engineering but also *restructuring* (the process of creating a logically equivalent system at the same level of abstraction, *eg.* transforming "spaghetti" code into structured code), and *re-engineering* (examination and alteration of a system to rebuild it in a new form and subsequent implementation of the new form).

3 Code slicing

Conceptually, reverse software specification is the converse of forward software reification. So it might suffice to define a formal semantics of the target language, apply it to the legacy code and submit the outcome to formal reasoning. In real situations, however, this approach is too naïve. In fact, the formal description of a large piece of code produced in an 'ad hoc' way (vulg. "spaghetti" code) can be overwhelmingly complex and counterproductive to inspect [2].

A similar kind of difficulty has been felt in informal software development and maintenance. Program *slicing*, introduced by Weiser [26], is known to help programmers in understanding foreign code and in debugging. A slice captures all computation on a given variable. This gives maintainers a straightforward technique for determining those statements and variables that may be modified and those that may not.

Can slicing techniques help in the application of denotational semantics techniques to formal reverse specification by calculation? We believe this is true and present an example of how it can help. But, before let us have a look at the foundations of the approach.

4 Denotational semantics vs program synthesis / analysis

Let P be a piece of algorithmic code and $[\![P]\!]$ denote the denotational semantics of P, ie, the input/output relation which captures the behaviour of P. In many cases, such a relation is a function which maps the *state* of P (*ie.* the set of all variables which P has access to) before execution takes place, to the state after such an execution.

In forward engineering one starts from a specification S and rewrites it over and over again,

$$S \supseteq S_1 \supseteq \ldots \supseteq S_n = [\![P]\!] \qquad (1)$$

until the semantics S_n of some piece of code P is found. Structured programming makes it possible to take advantage of the compositional properties of the available program combinators, *eg.* sequential composition

$$[\![P; Q]\!] = [\![Q]\!] \cdot [\![P]\!]$$

where "$\cdot$" denotes relational composition. So, if a specification S is written as $S = R \cdot T$ and we find that $R \supseteq [\![Q]\!]$ and $T \supseteq [\![P]\!]$ holds, then $S \supseteq [\![P; Q]\!]$ is a valid refinement step in program synthesis [3].

By contrast, it seems natural that program analysis should go the other way round: try and identify sequential chunks of code such that their semantics can be inferred and abstracted upon. There is a fact to retain, though: in going backwards along the $\supseteq$ direction in (1) one can add arbitrary nondeterminism and end up in a specification S which is too vague. In the limit, the universal relation is the reverse specification if any program, and this is clearly undesirable.

In this paper we deliberately ignore this aspect of the problem, by deciding to reverse specify a piece of code P only in terms of its functional specification. So we shall be dealing with functions and functional equality (rather than relations and the subset ordering) [4] and will resort to the HASKELL programming notation [5] to express the semantics of (deterministic) code.

[2]In many situations, the original code may be lost and only the binary code be available. In this case, the whole process must be preceded by *decompilation* [13].

[3]Obviously, other combinators such as `if-then-else`, `while` *etc.* exhibit similar semantic rules.

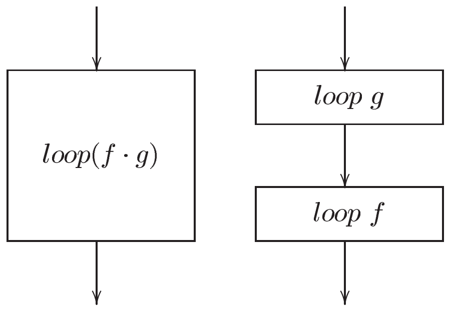
[4]Of course, we have to restrict ourselves to programs whose combinators do not involve nondeterminism.

5 Loop inter-combination, code fusion and slicing

When writing code, programmers tend to combine into a single programming construct (*eg.* a loop) two or more logically independent computations. This "trick" for efficiency can be expressed informally as follows,

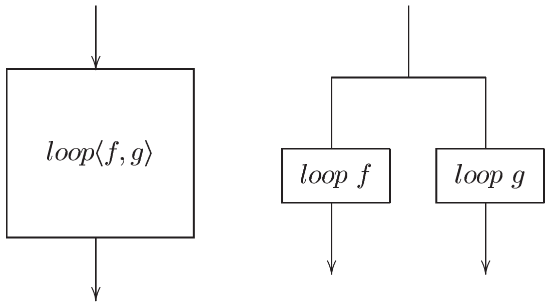
$$loop(f \cdot g) \quad \text{preferred to} \quad (loop\ f) \cdot (loop\ g) \qquad (2)$$

cf. the following drawing:



Programmers always rush for efficiency and so the same procedure is likely to be again applied to $f \cdot g$ itself, and so and so on.

In other situations the idea in the programmer's mind is to perform the computation of several (output, possibly independent) variables in the same loop, all sharing the same visit to the input data structure. This is depicted (for two such variables) in the following drawing,



— *ie.*

$$loop\langle f,g\rangle \quad = \quad \langle loop\ f, loop\ g\rangle \qquad (3)$$

— where the angle brackets in $\langle f,g\rangle$ denote the "parallel" execution of computations f and g. Should these computations on the right be independent, the loop-body on the left can become really inscrutable if the programmer doesn't bother to interleave the statements of f with the statements of g in an arbitrary, not obvious way. This is where code slicing proves really useful in debugging practice.

6 Algebra of programming

Such a systematic obsession for efficiency of the average programmer is not only error-prone but also (always) has the negative impact of obfuscating the original program plan, making it harder and harder to understand by others. However, such intuitions are correct and are actually validated by *algebra of programming* laws [3] which programmers (may) ignore but feel *obvious* about the semantics of the underlying programming language [5].

For instance, programming "trick" (2) is an instance of a class of formal program transformations known as *fusion laws*: two sequential computations of the same kind — in this case, *loop f* and *loop g* — are merged together (*ie.* "fused") in a single computation of the same kind — $loop(f \cdot g)$ in this case. Recall that we have decided to restrict ourselves to functional code blocks, *ie.*, pieces of code whose semantics can be expressed by functions f, g *etc.* So "$\cdot$" in $f \cdot g$ denotes function composition

$$\frac{B \xleftarrow{f} C \quad , \quad C \xleftarrow{g} A}{B \xleftarrow{f \cdot g} A}$$

such that

$$(f \cdot g)\ a \quad = \quad f(g\ a) \qquad (4)$$

On the other hand, programming *trick* (3) has to do with "fusing" two "parallel" computations into a single one and affiliates to another group of laws having to do with mutual recursion. These involve the "angle bracket" combinator of (3), which we will refer to as the "split" combinator:

$$\begin{array}{rcl} \langle f,g\rangle & : & A \longrightarrow B \times C \\ \langle f,g\rangle\ x & \stackrel{\text{def}}{=} & (f\ x, g\ x) \end{array} \qquad (5)$$

This is Backus [2] *construction* operator, which can be directly implemented in a high-level declarative language such as Haskell,

```
split :: (a -> b) -> (a -> c) -> a -> (b,c)
split f g x = (f x, g x)
```

[5] Side comment: what becomes far less obvious (to others) is their use in an in-discriminated and undocumented way!

where `(b,c)` means $B \times C$.

These combinators are rich in algebraic properties. For instance, the $\times$-*fusion law*

$$\langle g, h \rangle \cdot f = \langle g \cdot f, h \cdot f \rangle \quad (6)$$

expresses "distribution" of composition of over *split*, a law in which two "parallel" *consumer* functions g and h fuse with another, *producer* function f. This law is expressively depicted in a diagram,

$$\begin{tikzcd} B & B \times C \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & C \\ & A \arrow[ul, "g"] \arrow[u, "\langle g,h \rangle"] \arrow[ur, "h"'] & \\ & D \arrow[u, "f"] \arrow[uul, "g \cdot f"] \arrow[uur, "h \cdot f"'] & \end{tikzcd}$$

where *projections* π_1 and π_2 are as follows [6]:

$$\pi_1(a, b) \stackrel{\text{def}}{=} a \quad , \quad \pi_2(a, b) \stackrel{\text{def}}{=} b \quad (7)$$

Already known since Backus' *algebra of programs* [7], this *fusion-law* and many others form today a solid calculus which has reached the textbook format in [3].

7 Our work-plan

It is appealing to use all such laws — in the *opposite* direction, of course — to synthesize abstract specifications of ("spaghetti") code — that is, to reconstruct the original program plan. This is what motivates the present paper, in which code slicing plays the pragmatic role of unraveling independent parallel computations — as it is the case in debugging.

In fact, the lesson learnt from the code slicing community points to a clear direction in program understanding and debugging: instead of working with a monolithic state vector involving n state variables $(v_1, \ldots, v_n) :: V$ (for $V = V_1 \times \ldots \times V_n$, where V_i are available datatypes) and expressing all semantic rules in terms of state-vector transformations $f :: V \longleftarrow V$, one can "split" the effect on the state vector in terms of n independent computations $f_i :: V_i \longleftarrow V$ so that $f = \langle f_1, \ldots, f_n \rangle$. Each individual f_i is self-sufficient and smaller than the original code, therefore easier to reverse calculate.

[6]In HASKELL, π_1 is written `fst` and π_2 is written `snd`.

[7]John Backus [2] was among the first to alert computer programmers that computer languages alone are insufficient, and that only languages which exhibit an *algebra* for reasoning about the objects they purport to describe will be useful in the long run.

A formal way to obtain this "splitting" effect is based on the *mutual recursion law*, a standard result in the algebra of programming which expresses the fact that a function delivering a vector of n results can always be transformed into a "split" of n mutually dependent functions [8]. So — in a sense — reverse specification in this context means *re-introduction* of mutual recursion.

Of course, the more elaborate the function of unravel the harder to perform the calculation. The idea of using slicing is that of "shortcutting" part of this work via the syntactic separation of the program in its constituent slices. However, how sound is this strategy? How do we guarantee that, altogether, the slices' semantics "re-constitute" the whole program semantics? We have to formulate the following conjecture:

> **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, *ie.*
>
> $$[\![P]\!] = \langle [\![P_1]\!], \ldots, [\![P_n]\!] \rangle$$
>
> In other words, slicing is a semantically sound code-decomposition technique.

The relevance of this conjecture can only be appreciated in a formal context. In fact, most slicing techniques reported in the literature are "syntactic" in nature and/or rely on auxiliary data-flow analysis. We will rely on this conjecture, but its proof — which can only be carried out if the slicing technique is formally specified on a denotational semantics framework — is outside of the scope of this paper.

In general, the semantics of a program block P involving `while`, `for` or `do` statements and recursion will have the be inductively defined over some input type T, *eg.* a finite list, an array, a tree. So this type should be singled out from the space vector:

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

Slicing will supply us with a collection of slices P_i

$$[\![P_i]\!] :: V_i \longleftarrow T \times V_i \quad (8)$$

which we want to abstract into inductive functions with shape

$$f_i :: V_i \longleftarrow T \quad (9)$$

[8]With no loss in generality, this law will be presented, for $n = 2$, later on (section 8).

The transformation from (9) to (8) is a well-known technique for improving the efficiency of functional programs called *accumulation parameter introduction* [9].

Here we shall be interested in the reverse application of this rule, *ie.* we want to remove accumulations. Before presenting an example of such reverse transformations we need to dwell for a brief while on the algebra of *inductive* programming.

8 Algebra of programming (continued)

Inductive datatypes have a rich algebra which cannot be covered in this paper. With no loss in generality, we will focus on the datatype of *finite lists* (which is the one present in our illustration later on) and mention only the laws which are relevant for our calculations [10].

Consider the following inductive definition (in HASKELL) of the function which computes the sum of a list of elements of some numeric type `a`:

```
sum :: Num a => [a] -> a
sum [] = 0
sum (a:l) = a + sum l
```

In general, any list processing function with signature $B \xleftarrow{f} A^\star$ can be written according to the following recursive scheme, in HASKELL:

```
f [] = k
f (a:l) = g(a, f l)
```
(10)

for some $k \in B$ and $B \xleftarrow{g} A \times B$. For instance, $k = 0$ and $g(a, b) = a + b$ in *sum*.

A standard result in inductive datatype theory tells us that each instance of f is uniquely determined by the pair (k, g). Pairs of this kind are called *algebras* (= collections of functions) and can be described in a compact way by resorting to a combinator which *dualizes* split:

$$\begin{array}{rcl} [f,g] & : & A + B \longrightarrow C \\ [f,g]\ x & \stackrel{\text{def}}{=} & \begin{cases} x = i_1\ a & \Rightarrow\ f\ a \\ x = i_2\ b & \Rightarrow\ g\ b \end{cases} \end{array} \quad (11)$$

cf. diagram

$$\begin{tikzcd} A \arrow[r, "i_1"] \arrow[dr, "f"'] & A + B \arrow[d, "{[f,g]}"] & B \arrow[l, "i_2"'] \arrow[dl, "g"] \\ & C & \end{tikzcd} \quad (12)$$

where $A + B$ denotes the disjoint union of A and B which, in HASKELL, is supported directly via

```
data Either a b = Left a | Right b
```

(So the injections i_1 and i_2 become constructors `Left` and `Right`, respectively.)

Split and its dual are related to each other by the following *exchange law*, which enables us to express every function of type $B \times D \longleftarrow A + C$ in two alternative ways:

$$[\langle f, g\rangle, \langle h, k\rangle] \quad = \quad \langle [f, h], [g, k]\rangle \quad (13)$$

given functions $B \xleftarrow{f} A$, $D \xleftarrow{g} A$, $B \xleftarrow{h} C$ and $D \xleftarrow{k} C$.

Thanks to this new combinator one can record the whole information about *algebra* (k, g) above into a single arrow

$$B \xleftarrow{[\underline{k},g]} 1 + A \times B \quad (14)$$

where 1 denotes HASKELL's singleton type `()` and $\underline{k}$ denotes the "everywhere k" constant function. Going further in the same direction, we can let arrow (14) participate in a larger diagram which records the whole information about f (10):

$$\begin{tikzcd} A^\star \arrow[d, "f"'] & 1 + A \times A^\star \arrow[l, "in"'] \arrow[d, "id+id\times f"] \\ B & 1 + A \times B \arrow[l, "{[\underline{k},g]}"] \end{tikzcd} \quad (15)$$

In this diagram: $A^\star$ denotes the HASKELL type of finite lists `[A]`; *in* is the algebra $[\underline{[\,]}, (:)]$ which builds $A^\star$-lists [11]; *id* is the identity function such that $f \cdot id = id \cdot f = f$ for every f; the "recursive call" $id + id \times f$ involves the "product" combinator,

$$f \times g \quad \stackrel{\text{def}}{=} \quad \langle f \cdot \pi_1, g \cdot \pi_2\rangle \quad (16)$$

and its dual, the "sum" combinator:

$$f + g \quad \stackrel{\text{def}}{=} \quad [i_1 \cdot f, i_2 \cdot g] \quad (17)$$

[9]See *eg.* exercise 3.45 in [3].

[10]For a comprehensive account see reference [3] or, for an introduction, reference [15].

[11]With some abuse of notation, though: in HASKELL, constructor (:) is provided in curried form.

Diagram (15) expresses an equation about f

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

which we re-write into

$$f \cdot in = \alpha \cdot \mathsf{F}\ f \tag{18}$$

by introducing $\alpha = [\underline{k}, g]$ and $\mathsf{F}\ f = id + id \times f$. In fact, this equation is all we need to know to define f — provided we instantiate α, *ie.* k and g. To express this uniqueness of f, dependent on α, we write $(\!|\alpha|\!)$ — read "α-catamorphism" — instead of f:

$$(\!|\alpha|\!) \cdot in = \alpha \cdot \mathsf{F}\ (\!|\alpha|\!) \tag{19}$$

As an exercise, the reader can recover our starting definition for f (10) from this equation by applying standard laws of the algebra of programming known as $+$-*fusion*,

$$f \cdot [g, h] = [f \cdot g, f \cdot h] \tag{20}$$

and $+$-*absorption*

$$[g, h] \cdot (i + j) = [g \cdot i, h \cdot j] \tag{21}$$

among others.

We close this (very sketchy) account of the algebra of programming by presenting a law which will play a major role in the example to follow: the *mutual-recursion law*, also called "Fokkinga law":

$$\begin{cases} f \cdot in = h \cdot \mathsf{F}\ \langle f, g\rangle \\ g \cdot in = k \cdot \mathsf{F}\ \langle f, g\rangle \end{cases} \equiv \quad \langle f, g\rangle = (\!|\langle h, k\rangle|\!) \tag{22}$$

cf. diagrams

$$\begin{tikzcd}
\mathsf{T} \arrow[d, "f"'] & \mathsf{F}\ \mathsf{T} \arrow[l, "in"'] \arrow[d, "\mathsf{F}\ \langle f,g\rangle"] \\
A & \mathsf{F}\ (A \times B) \arrow[l, "h"]
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathsf{T} \arrow[d, "g"'] & \mathsf{F}\ \mathsf{T} \arrow[l, "in"'] \arrow[d, "\mathsf{F}\ \langle f,g\rangle"] \\
B & \mathsf{F}\ (A \times B) \arrow[l, "k"]
\end{tikzcd}$$

and

$$\begin{tikzcd}
\mathsf{T} \arrow[d, "\langle f,g\rangle"'] & \mathsf{F}\ \mathsf{T} \arrow[l, "in"'] \arrow[d, "\mathsf{F}\ \langle f,g\rangle"] \\
A \times B & \mathsf{F}\ (A \times B) \arrow[l, "\langle h,k\rangle"]
\end{tikzcd}$$

This law provides us with a very useful tool for "parallel loop" inter-combination.

9 Example — The WordCount Program

This example is taken from [6]. The starting point is the following source code:

```
#define YES 1
#define NO 0
main()
{
int c, nl, nw, nc, inword ;
inword = NO ;
nl = 0;
nw = 0;
nc = 0;
c = getchar();
while ( c != EOF ) {
    nc = nc + 1;
    if ( c == '\n')
    nl = nl + 1;
    if ( c == ' ' || c == '\n' || c == '\t')
          inword = NO;
    else if (inword == NO) {
          inword = YES ;
          nw = nw + 1;
    }
    c = getchar();
}
printf("%d",nl);
printf("%d",nw);
printf("%d",nc);
}
```

This can be recognized as a simplified version of the Unix `wc` command, which prints the number of bytes, words, and lines in files.

The HASKELL programming language [5] will be adopted to express the functional semantics of the slices of this piece of code. For space economy we will rely on the reader's intuition, rather than presenting the (somewhat) tedious process of synthesizing such semantics from the code itself. (See *eg.* [17] for a formal semantics of while-loops.)

9.1 Slicing

Gallagher and Lyle [6] identify the following slice decomposition lattice for this program:

$$\begin{tikzcd}
S(nc) \arrow[drr, dash] & S(nl) \arrow[dd, dash] & & S(nw) \arrow[dl, dash] \\
& & S(inword) \arrow[dl, dash] & \\
& S(c) & &
\end{tikzcd}$$

Clearly, there are 3 maximal slices to be extracted: $S(nc)$, $S(nl)$ and $S(nw)$. Slices $S(c)$ and $S(inword)$ are simpler, auxiliary slices which we analyse first.

The "bottom" slice is $S(c)$:

```
3 main()
4 {
5 int c ;
10 c = getchar();
11 while ( c != EOF ) {
21       c = getchar();
22 }
26 }
```

Because c is not an output variable, this slice will not become apparent in the specification to be inferred. Its major role is to unveil the underlying inductive type upon which the whole program is structured. Deliberately omitting the intricacies of the physical i/o instructions, one can infer that input is a finite sequence of characters. So we shall be dealing with catamorphisms over type $Char^\star$, all sharing inductive structure (15).

Next, slice $S(inword)$ is as follows:

```
1 #define YES 1
2 #define NO 0
3 main()
4 {
5 int c, inword ;
6 inword = NO ;
10 c = getchar();
11 while ( c != EOF ) {
15    if ( c == ' ' || c == '\n' || c == '\t')
16          inword = NO;
17    else if (inword == NO) {
18          inword = YES ;
20    }
21    c = getchar();
22 }
26 }
```

Variable *inword* supports the computation of *nw* but is not an output variable. Like c, it will not survive in the final specification. See below how we calculate its impact onto the semantics of $S(nw)$.

We proceed to extracting the semantics of the 3 maximal slices $S(nc)$, $S(nl)$ and $S(nw)$. Our convention is to denote the semantics of slice $S(v)$ of variable v by function v involving input list l, the one of which c is reading a value at time.

For $S(nc)$,

```
3 main()
4 {
5 int c, nc ;
9 nc = 0;
10 c = getchar();
11 while ( c != EOF ) {
12       nc = nc + 1;
20       }
21       c = getchar();
22 }
25 printf("%d",nc);
26 }
```

we infer the following semantics in HASKELL:

```
nc l = snd(nc_loop(l,0))

nc_loop ([],nc)   = ([],nc)
nc_loop (c:l,nc) = nc_loop(l,nc + 1)
```

Next, slice $S(nl)$

```
3 main()
4 {
5 int c, nl ;
7 nl = 0;
10 c = getchar();
11 while ( c != EOF ) {
13       if ( c == '\n')
14       nl = nl + 1;
21       c = getchar();
22 }
23 printf("%d",nl);
26 }
```

will exhibit the following semantics:

```
nl l = snd(nl_loop(l,0))

nl_loop ([],nl)  = ([],nl)
nl_loop (c:l,nl) =
        nl_loop(l, nl_aux(c,nl))
        where nl_aux(c,nl) | c == '\n' = nl + 1
                           | otherwise = nl
```

Finally, the semantics of $S(nw)$,

```
1 #define YES 1
2 #define NO 0
3 main()
4 {
5 int c, nw, inword ;
6 inword = NO ;
8 nw = 0;
10 c = getchar();
11 while ( c != EOF ) {
15     if ( c == ' ' || c == '\n' || c == '\t')
16           inword = NO;
```

```
    else if (inword == NO) {
          inword = YES ;
          nw = nw + 1;
    }
    c = getchar();
}
printf("%d",nw);
}
```

is somewhat more elaborate:

```
nw l = trd(nw_loop(l,False,0))
        where trd(_,_,x)=x

nw_loop ([],inword,nw)   = ([],inword,nw)
nw_loop (c:l,inword,nw) =
      if ( c == ' ' || c == '\n' || c == '\t')
      then nw_loop (l,False,nw)
      else if not inword
           then nw_loop (l, True, nw + 1)
           else nw_loop (l, inword,nw)
```

According to the conjecture of section 7, the whole program semantics is captured by

```
split3 nl nw nc
```

— *ie.* $\langle nl, nw, nc\rangle$ — where

```
split3 f g h x = (f x, g x, h x)
```

extends the *split* combinator (5) to three arguments.

9.2 Removing the accumulation parameters

For the technique of *accumulation parameter* introduction see *eg.* exercise 3.45 in [3]. Next we shall be interested in the reverse application of this rule. So our first task is to remove the accumulators (*ie.* the non-inductive parameters) of functions *nc*, *nl* and *nw*.

Concerning *nc*, we obtain

```
nc []  = 0
nc (c:l) = (succ . nc) l
```

where $succ\ n = n + 1$. This can be recognized as HASKELL's `length` function

$$length \quad = \quad (\!|[\underline{0}, succ\cdot\pi_2]|\!)$$

available from the language's Standard Prelude [5].

Next, *nl* is re-written into

```
nl []  = 0
nl (c:l) = if c == '\n'
           then (nl l) + 1
           else (nl l)
```

which can at once be identified as list catamorphism

$$(\!|[\underline{0}, (=\text{`}\backslash n\text{'})\cdot\pi_1 \rightarrow succ\cdot\pi_2, \pi_2]|\!)$$

where we use the *McCarthy's conditional* combinator defined by

$$p \rightarrow g, h \quad \overset{\text{def}}{=} \quad [g, h]\cdot p? \tag{23}$$

where

$$(p?)a \quad = \quad \begin{cases} p\ a & \Rightarrow \quad i_1\ a \\ \neg(p\ a) & \Rightarrow \quad i_2\ a \end{cases} \tag{24}$$

Finally, slice $S(nw)$ will require some preliminary work. First, an application of the *nestr* isomorphism

```
nestr(a,b,c) = (a,(b,c))
```

to the state space will isolate the accumulation parameter pair,

```
nw l = (snd . snd . nw_loop) (l,(False,0))
nw_loop ([],  p) = ([],p)
nw_loop (c:l,p) = nw_loop(l, nw_aux(c,p))
```

where

```
nw_aux(c,(inword, nw)) =
           if sep c
           then (False,nw)
           else if not inword then (True,nw+1)
                                 else (inword,nw)
```

and

```
sep c = ( c == ' ' || c == '\n' || c == '\t')
```

Then we are ready to remove the accumulator:

```
nw = snd . nw_rec
      where nw_rec []    = (False,0)
            nw_rec (c:l) = nw_aux(c,nw_rec l)
```

Although better than its previous version based on *nw_loop*, *nw* is still defined in terms of an auxiliary function *nw_rec*. Our final calculations will take care of removing the *inword* parameter of *nw_aux* — the remaining evidence that our reverse specification is not yet sufficiently abstract.

9.3 Introduction of mutual recursion

Let us focus on function $Bool \times Int \xleftarrow{nw\_rec} Char^\star$, which can be expressed as list catamorphism $(\!|[\langle\underline{\text{FALSE}}, \underline{0}\rangle, nw\_aux]|\!)$. Because it delivers a pair of outputs, function *nw_aux* can be decomposed (after some `if-then-else` playing around) into a split,

$$nw\_aux = \langle nw\_aux1, nw\_aux2\rangle$$

where

```
nw_aux1(c,(inword, nw)) = not (sep c)
```

and

```
nw_aux2(c,(inword, nw)) =
     if sep c || inword then nw else nw + 1
```

Then we get:

$$\begin{aligned} nw\_rec &= (\![[\langle \underline{\textsc{False}}, \underline{0}\rangle, \langle nw\_aux1, nw\_aux2\rangle]]\!) \\ &= \qquad \{ \text{ exchange law (13)} \} \\ &\qquad (\![\langle [\underline{\textsc{False}}, nw\_aux1], [\underline{0}, nw\_aux2]\rangle]\!) \end{aligned}$$

So nw_rec is in the situation of being handled by the *mutual-recursion law* (22), for $h = [\underline{\textsc{False}}, nw\_aux1]$ and $k = [\underline{0}, nw\_aux2]$. We will obtain

```
f []    = False
f (c:l) = not (sep c)

g []    = 0
g (c:l) = if (sep c || f l)
           then g l
           else g l + 1
```

Note that $nw = \pi_2 \cdot nw\_rec = \pi_2 \cdot \langle f, g\rangle = g$. So we rename g to nw, and — for improved readability — introduce $lookahead\_sep = \neg \cdot f$, *ie.*

```
lookahead_sep []    = True
lookahead_sep (c:l) = sep c
```

which enables us to swap the `if-then-else` and get a more intuitive reading of the final outcome of our reverse calculation:

```
nw []    = 0
nw (c:l) = if not (sep c) && lookahead_sep l
            then nw l + 1 else nw l
```

As anticipated earlier on, variable *inword* has disappeared throughout this calculation. In fact, it can be regarded as a state "flag" implementing the "separator lookahead" of function *lookahead_sep*.

Function *nw* is now a proper inductive function: it belongs to the class of so-called *paramorphisms* [12], which are very common in formal specification (*eg.* the usual definition of the factorial function $n!$ is a paramorphism). We can stop and feel happy about the level of abstraction of the outcome of the whole exercise.

10 Summary

We have combined a formal method — *algebra of programming* [3] — with a semi-formal one — *code slicing* [26] — in order to perform the reverse specification of a little program already studied in the code slicing literature: the WordCount program of [6]. We claim that we have gone deeper than [6] in understanding this piece of code.

A strong point of our approach is its constructive style, based on powerful algebraic laws of programming. This can go as far as imploding auxiliary variables and introducing mutual recursion, a specification mechanism which programmers never make explicit because they fear lacking efficiency. Our example is interesting also because it transforms all the semantic functions into well-known inductive schemata: cata/paramorphisms.

Slicing helps in trimming down the complexity of handling all program variables at the same time. In fact, the *mutual-recursion law* (22) could have been applied to the whole program — rather than to its slices — at the sacrifice of a lot more reasoning showing eventually that the three slices are independent of each other: a result known as the *banana-split law*,

$$\langle (\![i]\!), (\![j]\!)\rangle = (\![(i \times j) \cdot \langle \mathsf{F}\ \pi_1, \mathsf{F}\ \pi_2\rangle]\!) \qquad (25)$$

which is a special case of (22).

However, the application of slicing in this context is still dependent on its correctness, which should be properly dealt with on a denotational semantics setting.

11 Related work

In this section we briefly frame our work into ongoing research in this area.

References [4, 7, 8, 9] base their strategies for the (re)construction of specifications from legacy code on pre-/post-conditions.

In [11] the source code is translated into an intermediate language (UNIFORM), from which equational descriptions of the functionality are calculated. These are then expressed in an intermediate functional language. Finally, the objects sketched at the first stage are completed with the functional descriptions that define their operational behaviour.

In [25] the source code is reconstructed until a recursive version is reached, upon which properties of the program can be deduced which allow one to perform reductions. Data structures are treated until the references to concrete variables disappear. The outcome is an abstract program that can still be made simpler by means of transformations, until a high level abstract specification of is reached. (Our work clearly affiliates to this approach.)

Reference [14] shows how formalisms similar to those that we are trying to apply in this paper to the algorithmic code, have already been successfully applied to "data understanding" and the improvement of the quality of the data, *cf.* formal DRE (data reverse engineering).

The repertoire of formal techniques for reverse engineering further includes "type inference" [19, 10, 20, 21] and concept/cluster analysis [18]. These are applicable mainly to the detection of objects and can also be combined with techniques proposed in this paper. Likewise, semi-formal techniques [22, 23] for the detection of recurrent algorithmic structures can also contribute to the identification — and later to the formalization — of program patterns.

12 Future work

As pointed out in the introduction, the development of the technique for reverse engineering reported in this paper is work in progress and some of its problems are still open. At the heart of these we place the conjecture of section 7, whose proof (in a denotational semantics setting) is required before we regard slicing as a semantically sound code-decomposition technique.

A forthcoming master thesis [24] is expected to present several exercises such as WordCount which will let us to know more about which laws of programming are relevant in this context and to improve the interplay between the code slicing and the algebra of programming techniques.

It is our current belief that the slice decomposition lattices of [6] will play a significant role in structuring the overall method, as can already be seen in section 9.

On the formal side of the approach there is a lot to be done. In particular, many program structures will require laws more powerful than those which we have been thinking of — for instance, comonadic calculations [16]. For the moment, we are thinking in terms of examples which we can interpret functionally. By moving to *relation algebra* [1, 3] we expect to generalize our reverse calculations to any sliceable program P.

References

[1] R. C. Backhouse. Fixed point calculus, 2000. Summer School and Workshop on *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, Lincoln College, Oxford, UK 10th to 14th April 2000.

[2] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *CACM*, 21(8):613–639, August 1978.

[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] 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.

[5] Simon Peyton Jones (ed.), John Hughes (ed.), Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Report on the programming language Haskell 98 — a non-strict, purely functional language. Technical report, February 1999.

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

[7] 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.

[8] G.C. Gannod and B.H.C. Cheng. Using informal and formal techniques for the reverse engineering of C programs. Technical report, DCS, Michigan State University, 1996.

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

[10] Tobias Kuipers and Leon Moonen. Types and concept analysis for legacy systems. *8th IWPC'00, IEEE Computer Society*, 2000.

[11] K. Lano, P. Breuer, and H. Haughton. Reverse-engineering COBOL via formal methods. Technical Report TR-16-91, Oxford University, May 1991.

[12] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4:413–424, 1992.

[13] A. Mycroft. Type-based decompilation. In S. D. Swierstra, editor, *ESOP'99 - European Symposium On Programming*, Lecture Notes in Computer Science. Springer, 1999.

[14] 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.

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

[16] A. Pardo. Towards merging recursion and comonads. In *WGP'2000, Ponte de Lima*, July 2000.

[17] Paul Taylor. An exact interpretation of **while**. In *Theory and Formal Methods 1993: Proceedings of the First Imperial College, Department of Computing, Workshop on Theory and Formal Methods*, 1993.

[18] Arie van Deursen and Tobias Kuipers. Identifying objects using cluster and concept analysis. *ICSE'99, ACM*, 1999.

[19] Arie van Deursen and Leon Moonen. Type inference for cobol systems. *5th WCRE'98, IEEE Computer Society*, 1998.

[20] Arie van Deursen and Leon Moonen. Understanding cobol system using types. *In Proceedings 7th Int. Workshop on Program Comprehension, IWPC'99*, pages 74–83, 1999.

[21] Arie van Deursen and Leon Moonen. Understanding cobol systems using inferred types. *7th IWPC'99, IEEE Computer Society*, 1999.

[22] G. Villacicencio. Program analysis for the construction of libraries of programming plans applying slicing. In *XIV Brazilian Symposium on Software Engineering Promoted by SBC, Brazilian Computer Society, João Pessoa – Paraíba – Brasil, October*, October 4-6th 2000.

[23] G. Villacicencio. Program analysis for the automatic detection of programming plans applying slicing. In *5th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, March 2001.

[24] G. Villavicencio. Formalización de una estrategia integral de ingeniería reversa, 2001. Master's thesis in preparation (University of San Luis, Argentina).

[25] M. P. Ward. Program analysis by program transformation. *The Computer Journal*, 1995.

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