*Journal of Universal Computer Science, vol. 7, no. 8 (2001), 754-781*
*submitted: 18/5/01, accepted: 21/8/01, appeared: 28/8/01 © Springer Pub. Co.*

# "Bagatelle in C arranged for VDM SoLo"

José N. Oliveira
(Universidade do Minho, Braga, Portugal
Senior R&D Partner, SIDEREUS S.A., Porto, Portugal
jno@di.uminho.pt)

**Abstract:** This paper sketches a reverse engineering discipline which combines formal and semi-formal methods. Central to the former is denotational semantics, expressed in the ISO/IEC 13817-1 standard specification language (VDM-SL). This is strengthened with *algebra of programming*, which is applied in "reverse order" so as to reconstruct formal specifications from legacy code. The latter include *code slicing*, a "shortcut" which trims down the complexity of handling the formal semantics of all program variables at the same time.

A key point of the approach is its constructive style. Reverse calculations go as far as absorbing auxiliary variables, introducing mutual recursion (if applicable) and reversing semantic denotations into standard generic programming schemata such as cata/paramorphisms.

The approach is illustrated for a small piece of code already studied in the code-slicing literature: Kernighan and Richtie's *word count* C programming "bagatelle".

**Key Words:** Reverse engineering; denotational semantics; slicing; algebra of programming.

**Category:** F.3.1 (Specifying and Verifying and Reasoning About Programs); D.2.1 (Requirements/Specifications)

## 1 Introduction

The development of computer software has always been a contradictory activity. Four decades ago, a "software crisis" was identified which drew attention to the poor technical sophistication of the (then emerging) software technology. Many years passed, the advent of home computing and world-wide network facilities has brought computer software unprecedented relevance in everybody's life. However, the theoretical advances in programming science — which has become a stable discipline meanwhile — are still by and large ignored by the ever growing community of programmers.

The situation is no better in education: software design remains among the very few topics which many engineering departments still accept to address with little (if any) scientific basis.

Time-to-market is among the main factors enforcing *ad hoc* programming practice. However, a quick market entails indirect costs such as permanent risk, poor quality and low reliability of the produced code, and hard maintenance.

In a situation in which the only *quality certificate* of the running software artifact still is life-cycle endurance, customers and software producers are little prepared to change, modify or improve running code. So there is little opportunity for brand new, technically sound code to replace the old one. (One is never sure about the implications of a software update.) However, faced with so risky a dependence on legacy software, managers are more and more prepared to spend resources to increase confidence on — *ie.* the level of *understanding* of — their (untouchable) code.

## 2 Program understanding

Program understanding affiliates to *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* [CI90]. Reverse engineering includes *reverse specification* — the analytical process of inferring the original specification (which may have never been written) of some running piece of software.

If (on statistical grounds) *forward* software engineering can today be regarded as a lost opportunity for formal methods [1], its converse still looks a promising area for their application. This is due to the complexity of reverse engineering problems. So it is likely that the formal techniques developed by academics for the production of fresh, high quality software will eventually see the light of (industrial) success in their *reverse application* to pre-existing code. Even if, for this purpose, they have to merge with other, informal program maintenance and debugging techniques.

## 3 About this paper

This paper is a modest contribution in the spirit of the last paragraph above. It reports 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 enriches standard *denotational semantics* techniques with the *algebra of programming* [BdM97], which is applied in "reverse order" so as to reconstruct the formal specifications of legacy code. Because of the complexity inherent in handling formal semantic descriptions of algorithmic code, reverse algebraic calculation is preceded by *code slicing* [Wei81].

This work is the follow up of a project which has addressed *data reverse engineering* (DRE) in a similar way [NSO99]: data are reversed by calculation according to an algebra of data-representation laws which include the transformation of relational database meta-data into abstract ISO/IEC 13817-1 standard (VDM-SL) formal descriptions [FL98].

Code reversal is, in general, harder than DRE and this paper will not present a general solution to the problems involved. Instead, a small example is worked out which triggers some intuitions about the strategy to follow in real situations. This example is the "word count" programming "bagatelle" [2] presented by Kernighan and Richtie's in their well-known introductory book to the C programming language (Fig. 1). This can

[1] With notable exceptions in areas such as *safety-critical* and *dependable computing*.
[2] In the musical terminology, a *bagatelle* ("musical trifle") is a short light or whimsical piece, usually written for piano.

```
#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);
}
```

**Figure 1:** The *word count* program [KR78].

be recognized as the core of the Unix wc command, which prints the number of bytes, words, and lines in files.

However simple, this example is expressive enough to illustrate our main point: that, in practice, denotational techniques alone are insufficient for code reversal, and that they benefit from combining with other — either formal or informal — methods.

The remainder of this paper is structured as follows: in the following section we introduce our (light-weight) approach to denotational semantics, which is illustrated in section 5 for the *word count* example, using VDM-SL notation. In sections 6 and 7 we motivate the application of slicing to formal semantics. This is illustrated in section 8. Section 9 motivates the need for *a posteriori* algebraic calculations, to be applied to each particular slice. The algebra of programming is briefly presented in sections 10 and 11, and finally applied to one of the slices of the word-count example (section 12). The paper ends with some conclusions and review of related work.

## 4 Denotational semantics for program synthesis / analysis

Let $P$ be a piece of algorithmic code and $[\![P]\!]$ denote its denotational semantics, ie, the input/output relation which captures the behaviour of $P$. In forward engineering one starts from a specification $S$ and rewrites it over and over again,

$$S \supseteq S_1 \supseteq \ldots \supseteq S_n = [\![P]\!] \tag{1}$$

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, for instance sequential composition

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

where "$\cdot$" denotes relational composition. So, if specification $S = R \cdot T$ is written down and one finds $Q$ and $P$ such that $R \supseteq [\![Q]\!]$ and $T \supseteq [\![P]\!]$ hold, then $S \supseteq [\![P; Q]\!]$ is a valid refinement step in program synthesis.

By contrast, it seems natural that program analysis should go the other way round: try and identify 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.

For $P$ a deterministic program, relation $[\![P]\!]$ specializes to 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. This retraction to a functional semantics, albeit restrictive in general, is acceptable in face of simple programs ("bagatelles") such as *word count* (Fig. 1). The theory gets simpler and more intuitive and this helps the (informal) reader to appreciate the power of formal reasoning. This explains our decision in this paper to reverse specify `main` in the *word-count* example only in terms of its functional specification [3]. So we shall be dealing with functions and functional equality, rather than relations and the subset ordering.

## 5 Starting denotational semantics for *word count*

The VDM-SL specification language [FL98] will be adopted to express the formal semantics of (deterministic) code such as *word count*. One starts by inferring the program state from the program variables:

[3] Later on we will address shortcomings of this decision, acceptable only in face of deterministic, terminating programs. The trade-off between the relational and functional foundations for program calculation is apparent in [BdM97]. The power of the relational approach can be appreciated in [BH93, Bac00], among a vast literature on this subject.

$$
\begin{aligned}
St1 :: \; & c : \mathsf{char} \\
& nl : \mathbb{Z} \\
& nw : \mathbb{Z} \\
& nc : \mathbb{Z} \\
& inword : \mathbb{B}
\end{aligned}
$$

For notation convenience, each selector in $St1$ is named after the corresponding program variable. Note the char and $\mathbb{B}$ abstractions of $\mathbb{Z}$ in variables $c$ and $inword$, respectively. Next, the input stream is made explicit in a way such that `EOF` is modelled by nil ,

$$
\begin{aligned}
St2 :: \; & stdin : \mathsf{char}^* \\
& c : [\mathsf{char}] \\
& nl : \mathbb{Z} \\
& nw : \mathbb{Z} \\
& nc : \mathbb{Z} \\
& inword : \mathbb{B}
\end{aligned}
$$

*cf.* the following semantics for $getchar$:

$$
\begin{aligned}
& getchar : St2 \to St2 \\
& getchar\,(\mathsf{mk}\text{-}St2\,(io, \text{-}, l, w, c, i)) \;\underline{\triangle} \\
& \quad \mathsf{if}\; io = [\,] \\
& \quad \mathsf{then}\; \mathsf{mk}\text{-}St2\,(io, \mathsf{nil}\,, l, w, c, i) \\
& \quad \mathsf{else}\; \mathsf{mk}\text{-}St2\,(\mathsf{tl}\; io, \mathsf{hd}\; io, l, w, c, i);
\end{aligned}
$$

Let $wcount$ be the function which captures the semantics of the whole program, that is

$$[\![\; \texttt{main()} ]\!] = wcount$$

In VDM-SL notation one has:

$$
\begin{aligned}
& wcount : \mathsf{char}^* \to \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \\
& wcount\,(l) \;\underline{\triangle} \\
& \quad \mathsf{let}\; s = wloop\,(\mathsf{mk}\text{-}St2\,(l, \mathsf{nil}\,, 0, 0, 0, \mathsf{false}))\; \mathsf{in} \\
& \quad \mathsf{mk}\text{-}\,(s.nl, s.nw, s.nc);
\end{aligned}
$$

The while-loop in `main()` is modelled by auxiliary function $wloop$, which updates the state, of type $St2$. The loop initialization is captured by record

$$\mathsf{mk}\text{-}St2\,(l, \mathsf{nil}\,, 0, 0, 0, \mathsf{false})$$

passed as parameter to $wloop$:

$$
\begin{array}{l}
wloop : St2 \to St2 \\
wloop\,(s) \triangleq \\
\quad \text{let } r = getchar\,(s) \text{ in} \\
\quad \text{if } r.c = \text{nil} \\
\quad \text{then } r \\
\quad \text{else let } c' = r.c, \\
\qquad\qquad nc' = r.nc + 1, \\
\qquad\qquad nl' = \text{if } c' = \text{'\textbackslash}n\text{'} \\
\qquad\qquad\qquad \text{then } r.nl + 1 \\
\qquad\qquad\qquad \text{else } r.nl, \\
\qquad\qquad nw' = \text{if } (c' = \text{' '} \lor c' = \text{'\textbackslash}n\text{'} \lor c' = \text{'\textbackslash}t\text{'}) \\
\qquad\qquad\qquad \text{then } r.nw \\
\qquad\qquad\qquad \text{else if } \neg\,(r.inword) \\
\qquad\qquad\qquad\quad \text{then } r.nw + 1 \\
\qquad\qquad\qquad\quad \text{else } r.nw, \\
\qquad\qquad in' = \text{if } (c' = \text{' '} \lor c' = \text{'\textbackslash}n\text{'} \lor c' = \text{'\textbackslash}t\text{'}) \\
\qquad\qquad\qquad \text{then false} \\
\qquad\qquad\qquad \text{else if } \neg\,(r.inword) \\
\qquad\qquad\qquad\quad \text{then true} \\
\qquad\qquad\qquad\quad \text{else } r.inword \text{ in} \\
\qquad wloop\,(\text{mk-}St2\,(r.stdin, c', nl', nw', nc', in'));
\end{array}
$$

State model $St2$ can be made simpler by filtering the redundancy of entries $c$ and $stdin$. This means to data-abstract $St2$ to

$$
\begin{array}{l}
St :: stdin : \text{char}^* \\
\qquad nl : \mathbb{Z} \\
\qquad nw : \mathbb{Z} \\
\qquad nc : \mathbb{Z} \\
\qquad inword : \mathbb{B}
\end{array}
$$

under abstraction (*retrieve* [Jon86]) function

$$
\begin{array}{l}
absSt : St2 \to St \\
absSt\,(\text{mk-}St2\,(s, c, l, w, n, i)) \triangleq \\
\quad \text{if } c = \text{nil} \\
\quad \text{then mk-}St\,([], l, w, n, i) \\
\quad \text{else mk-}St\,([c] \frown s, l, w, n, i)
\end{array}
$$

Then $getchar$ reduces to (hd , tl )-manipulation, leading to a simpler version of $wloop$:

$wloop$ : $St \rightarrow St$
$wloop\,(\text{mk-}St\,(s, l, w, c, i)) \triangleq$
  if $s = []$
  then mk-$St\,(s, l, w, c, i)$
  else let $w' =$ if $sep\,(\text{hd}\ s)$
                then $w$
                else if $\neg\, i$
                     then $w + 1$
                     else $w$,
        $i' =$ if $sep\,(\text{hd}\ s)$
               then false
               else if $\neg\, i$
                    then true
                    else $i$ in
       $wloop\,(\text{mk-}St\,(\text{tl}\ s,$
                  if hd $s =$ '$\backslash n$'
                  then $l + 1$
                  else $l$,
                  $w', c + 1, i'))$;

where an auxiliary function was introduced

$sep$ : char $\rightarrow \mathbb{B}$
$sep\,(c) \triangleq$
  $c =$ ' ' $\vee\ c =$ '$\backslash n$' $\vee\ c =$ '$\backslash t$';

abstracting the test for a separator character. Finally, $wcount$ is redefined accordingly and renamed to $wc$:

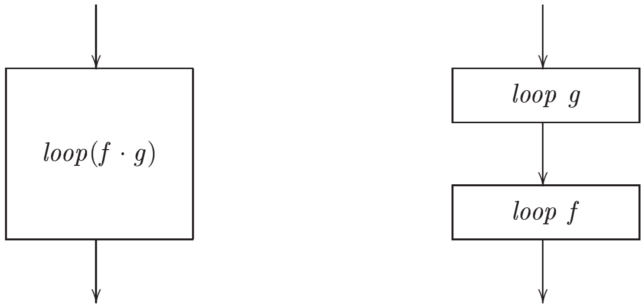
$wc$ : char$^*$ $\rightarrow \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$
$wc\,(l) \triangleq$
  let $s = wloop\,(\text{mk-}St\,(l, 0, 0, 0, \text{false}))$ in
  mk- $(s.nl, s.nw, s.nc)$

## 6 Loop inter-combination and code fusion

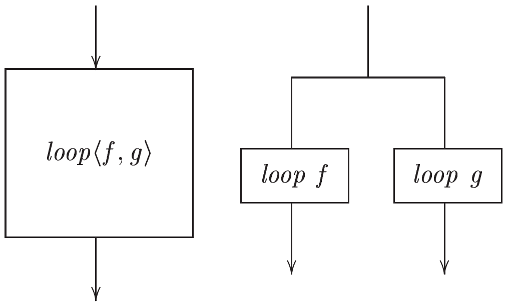
When writing code such as *word count*, programmers tend to combine into a single programming construct (*eg.* a loop) two or more logically independent computations. This saves auxiliary (*eg.* intermediate) data-structuring and the programming "tricks" to achieve efficiency in this way can be framed into two broad categories, one "sequential" and one "parallel". The former can be expressed informally as follows,

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

as illustrated in a diagram:




In other situations the idea in the programmer's mind is to perform the computation of several (possibly independent) output 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,




that is,

$$loop\langle f, g\rangle \qquad \text{preferred to} \qquad \langle loop\ f, loop\ g\rangle \tag{3}$$

where the angle brackets in $\langle f, g\rangle$ denote the "parallel" execution of computations $f$ and $g$ on the same input.

The *word count* program of Fig. 1 is an instance of situation (3): the computations of `nc`, `nl` and `nw` (resp. the number of characters, number of lines and number of words in the input stream) proceed in parallel, *ie.* in the same `while`-loop access to the input stream.

## 7 Code slicing

As happens with *word count*, the loop-intercombination strategy suggested in (3) may involve computations on the right which are independent of each other. 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 useful in debugging practice. Program *slicing* was introduced by Weiser [Wei81] as a means to help programmers in understanding foreign code and in debugging. It is a technique for restricting the behaviour of a program $P$ to some fragment of interest. The (decomposition) slice $S(x) \subseteq P$ on variable $x$ yields that portion of the program which contributes to the computation of $x$. Slices are executable (sub)programs which can be extracted automatically from the source code by data flow and control flow analysis. Gallagher and Lyle [GL91] show how such slices, ordered by set inclusion, form a semi-lattice where *meet* corresponds to code sharing.

The lesson learned from the code slicing community points to a clear direction in program understanding: instead of working with a monolithic state vector involving $n$ state variables $\langle v_1, \ldots, v_n \rangle : S$, for $S = V_1 \times \ldots \times V_n$ and each $v_i$ of type $V_i$, and expressing semantics in terms of state-vector transformations $f : S \longleftarrow S$, one "splits" the effect on the state vector in terms of $n$ independent computations $f_i : V_i \longleftarrow S$. Each individual $f_i$ is self-sufficient and smaller than the original code, therefore easier to understand (or reverse calculate).

The idea of using slicing is that of "short-cutting" the work of understanding a large program via the syntactic separation of the program in its constituent slices. However, how sound is this strategy? On a denotational semantics basis, how do we guarantee that, altogether, the slices' semantics "re-constitute" the whole program semantics? We will rely on the following 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.*
>
> $$[\![P]\!] = \langle [\![P_1]\!], \ldots, [\![P_n]\!] \rangle \tag{4}$$
>
> In other words, slicing is a semantically sound code-decomposition technique.

The angle-bracket combinator of (4) will be formalized in section 11. Although the equality does not hold for all programs (see section 15), it does hold for a broad class of deterministic programs which includes `wc`.

## 8 The slices of `wc` and their formal semantics

In general, the semantics of a program block $P$ involving `while`, `for` or `do` statements and/or recursion will have to 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:

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

Slicing will supply a collection of slices $P_i$ such that

$$[\![P_i]\!] : V_i \longleftarrow T \times V_i \tag{5}$$

which we may want to abstract into inductive functions with shape

$$f_i : V_i \longleftarrow T \tag{6}$$

The transformation from (6) to (5) is a well-known technique for improving the efficiency of functional programs called *accumulation parameter introduction* [BdM97]. Below we shall be interested in the reverse application of this rule, that is, we want to remove accumulations.

Let us instantiate this process for *word count* (Fig. 1). For this piece of code, Gallagher and Lyle [GL91] identify the following slice decomposition semi-lattice:

$$\begin{tikzcd}
S(nc) & S(nl) & & S(nw) \\
& & S(inword) \arrow[ur, dash] & \\
& S(c) \arrow[uul, dash] \arrow[uu, dash] \arrow[ur, dash] & &
\end{tikzcd}$$

Clearly, there are 3 maximal slices to be extracted

$$\begin{cases} S(nc) \text{ — number of characters} \\ S(nl) \text{ — number of lines} \\ S(nw) \text{ — number of words} \end{cases}$$

which match with the triple output of the program. Let us analyse each of these. The character count slice $S(nc)$ is

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

with semantics captured by function

$nc : \mathsf{char}^* \to \mathbb{Z}$
$nc\,(l) \triangleq$
$\quad ncloop\,(\mathsf{mk}\text{-}Stnc\,(l, 0)).nc;$

which calls

$$
\begin{array}{l}
ncloop : Stnc \to Stnc \\
ncloop\,(\mathsf{mk}\text{-}Stnc\,(s, c)) \ \underline{\triangle} \\
\quad \mathsf{if}\ s = [] \\
\quad \mathsf{then}\ \mathsf{mk}\text{-}Stnc\,(s, c) \\
\quad \mathsf{else}\ ncloop\,(\mathsf{mk}\text{-}Stnc\,(\mathsf{tl}\ s, c + 1));
\end{array}
$$

defined over the $nc$-projection of $St$:

$$
\begin{array}{l}
Stnc :: stdin : \mathsf{char}^* \\
\qquad\ \ nc : \mathbb{Z}
\end{array}
$$

Entry $nc$ of $Stnc$ is clearly an *accumulation parameter* in function $nc$, whose removal is the inverse of a standard program transformation (exercise 3.45 in [BdM97]). We thus “get rid of the state” and obtain

$$
\begin{array}{l}
nc : \mathsf{char}^* \to \mathbb{Z} \\
nc\,(l) \ \underline{\triangle} \\
\quad \mathsf{if}\ l = [] \\
\quad \mathsf{then}\ 0 \\
\quad \mathsf{else}\ nc\,(\mathsf{tl}\ l) + 1;
\end{array}
$$

easily recognizable as VDM-SL’s primitive “length” function. So we are done,

$$nc = \mathsf{len}$$

and can proceed to the line-count 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;
21       c = getchar();
22 }
23 printf("%d",nl);
26 }
```

This corresponds to sliced state

$$
\begin{array}{l}
Stnl :: stdin : \mathsf{char}^* \\
\qquad\ \ nl : \mathbb{Z}
\end{array}
$$

and exhibits semantics

$nl : \mathsf{char}^* \to \mathbb{Z}$
$nl\,(l) \triangleq$
$\quad nlloop\,(\mathsf{mk}\text{-}Stnl\,(l, 0)).nl;$
$nlloop : Stnl \to Stnl$
$nlloop\,(\mathsf{mk}\text{-}Stnl\,(s, l)) \triangleq$
$\quad \mathsf{if}\ s = []$
$\quad \mathsf{then}\ \mathsf{mk}\text{-}Stnl\,(s, l)$
$\quad \mathsf{else}\ nlloop\,(\mathsf{mk}\text{-}Stnl\,(\mathsf{tl}\ s, nlaux\,(\mathsf{hd}\ s, l)));$

for auxiliary function

$nlaux : \mathsf{char} \times \mathbb{Z} \to \mathbb{Z}$
$nlaux\,(c, i) \triangleq$
$\quad \mathsf{if}\ c = \text{'}\backslash n\text{'}$
$\quad \mathsf{then}\ i + 1$
$\quad \mathsf{else}\ i;$

Again the same standard transformation will lead to a state-free denotation:

$nl : \mathsf{char}^* \to \mathbb{Z}$
$nl\,(l) \triangleq$
$\quad \mathsf{if}\ l = []$
$\quad \mathsf{then}\ 0$
$\quad \mathsf{else\ if}\ (\mathsf{hd}\ l) = \text{'}\backslash n\text{'}$
$\qquad\quad \mathsf{then}\ nl\,(\mathsf{tl}\ l) + 1$
$\qquad\quad \mathsf{else}\ nl\,(\mathsf{tl}\ l);$

Finally, we have to cope with word counting slice $S(nw)$:

```
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;
17     else if (inword == NO) {
18            inword = YES ;
```

```
            ++nw;
      }
      c = getchar();
  }
  printf("%d",nw);
  }
```

Fact $S(nw) \supseteq S(inword)$ is apparent from the $nw$ projection of $St$

$$
\begin{aligned}
& Stnw :: stdin : \mathsf{char}^* \\
& \qquad\quad nw : \mathbb{Z} \\
& \qquad\quad inword : \mathbb{B}
\end{aligned}
$$

which includes $inword$, and from $[\![S(nw)]\!]$, which is function

$$
\begin{aligned}
& nw : \mathsf{char}^* \to \mathbb{Z} \\
& nw\,(l) \;\underline{\triangle} \\
& \quad nwloop\,(\textsf{mk-}Stnw\,(l, 0, \mathsf{false})).nw; \\
& nwloop : Stnw \to Stnw \\
& nwloop\,(\textsf{mk-}Stnw\,(s, w, i)) \;\underline{\triangle} \\
& \quad \mathsf{if}\ s = [\,] \\
& \quad \mathsf{then}\ \textsf{mk-}Stnw\,(s, w, i) \\
& \quad \mathsf{else\ let}\ w' = \mathsf{if}\ sep\,(\mathsf{hd}\ s) \\
& \qquad\qquad\qquad\quad \mathsf{then}\ w \\
& \qquad\qquad\qquad\quad \mathsf{else\ if}\ \neg\, i \\
& \qquad\qquad\qquad\qquad\quad \mathsf{then}\ w + 1 \\
& \qquad\qquad\qquad\qquad\quad \mathsf{else}\ w, \\
& \qquad\qquad\ i' = \mathsf{if}\ sep\,(\mathsf{hd}\ s) \\
& \qquad\qquad\qquad \mathsf{then\ false} \\
& \qquad\qquad\qquad \mathsf{else\ if}\ \neg\, i \\
& \qquad\qquad\qquad\qquad \mathsf{then\ true} \\
& \qquad\qquad\qquad\qquad \mathsf{else}\ i\ \mathsf{in} \\
& \qquad\quad nwloop\,(\textsf{mk-}Stnw\,(\mathsf{tl}\ s, w', i'));
\end{aligned}
$$

Clearly, $[\![S(nw)]\!]$ is less obvious than what we have been seen above for the other slices. Firstly, because there are two state variables ($wc$ and $inword$) instead of one. So our first step is to package them together,

$$
\begin{aligned}
& Stnw :: stdin : \mathsf{char}^* \\
& \qquad\quad acp : Acp \\
& Acp :: nw : \mathbb{Z} \\
& \qquad\ inword : \mathbb{B}
\end{aligned}
$$

and redefine $nwloop$ accordingly

$$\begin{array}{l} nwloop : Stnw \rightarrow Stnw \\ nwloop\,(\textsf{mk-}Stnw\,(s, p)) \triangleq \\ \quad \textsf{if } s = [] \\ \quad \textsf{then mk-}Stnw\,(s, p) \\ \quad \textsf{else } nwloop\,(\textsf{mk-}Stnw\,(\textsf{tl } s, nwaux\,(\textsf{hd } s, p))); \end{array}$$

where $nwaux$ is introduced to capture the state accumulation process:

$$\begin{array}{l} nwaux : \textsf{char} \times Acp \rightarrow Acp \\ nwaux\,(c, \textsf{mk-}Acp\,(w, i)) \triangleq \\ \quad \textsf{if } sep\,(c) \\ \quad \textsf{then mk-}Acp\,(w, \textsf{false}) \\ \quad \textsf{else if } \neg\, i \\ \qquad \textsf{then mk-}Acp\,(w + 1, \textsf{true}) \\ \qquad \textsf{else mk-}Acp\,(w, i); \end{array} \tag{7}$$

Secondly, because the removal of the $Acp$ accumulator will not bring — unlike the preceding examples — a denotation as clean as we would like:

$$\begin{array}{l} nw : \textsf{char}^* \rightarrow \mathbb{Z} \\ nw\,(l) \triangleq \\ \quad nwrec\,(l).nw; \\ nwrec : \textsf{char}^* \rightarrow Acp \\ nwrec\,(s) \triangleq \\ \quad \textsf{if } s = [] \\ \quad \textsf{then mk-}Acp\,(0, \textsf{false}) \\ \quad \textsf{else } nwaux\,(\textsf{hd } s, nwrec\,(\textsf{tl } s)); \end{array} \tag{8}$$

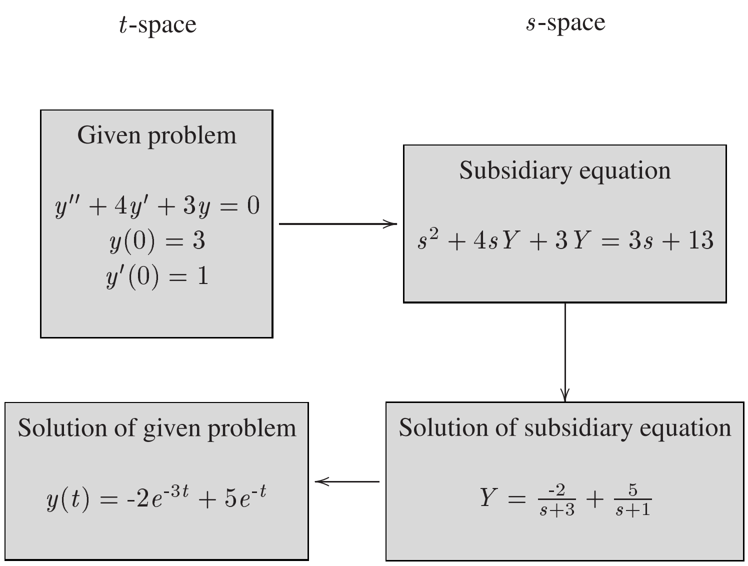
The fact that an auxiliary function ($nwrec$) is still needed brings about some questions: shouldn't Boolean entry $inword$ in $Acp$ have already disappeared? how do we proceed? is there a limit to the program transformations we have been applying so far?

## 9 Need for a notation transformation

Certainly we could go further in conventional (*ie.* lambda-calculus based) transformations. However, such "*bagatelle*-sized" reasoning is unlikely to scale up to realistic situations. Why? Legacy code normally involves intricate collections of global variables whose meaning is obfuscated by loop-intercombination, making it harder and harder to understand by others. It is no wonder that the formal semantics of code made efficient in this way leads to voluminous mathematical expressions involving large state vectors which resist to symbolic manipulation.

However, the programming intuitions implicit in (2) and (3) make sense and actually validated by *algebraic* laws [BdM97] which programmers (as a rule) ignore but feel *obvious* about the semantics of the underlying programming language [4].

Notation seems to be a factor against the explicit use of such laws in denotational semantics. Pointwise notation involving operators as well as variable symbols, logical connectives, quantifiers, *etc.* is not abstract enough. It also entails a loss in genericity which conventional engineering mathematics has learned to solve elsewhere by changing the "mathematical space", for instance by moving (temporarily) from the $t$-*space* to the $s$-*space* in the *Laplace transformation* [Kre88]:




Quoting [Kre88], p.242:

*The Laplace transformation is a method for solving differential equations (...) The process of solution consists of three main steps:*

**1st step.** *The given "hard" problem is transformed into a "simple" equation (*subsidiary equation*).*

**2nd step.** *The subsidiary equation is solved by* ***purely algebraic*** *manipulations.*

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

**3rd step.** *The solution of the subsidiary equation is transformed back to obtain the solution of the given problem.*

*In this way the Laplace transformation reduces the problem of solving a differential equation to an* ***algebraic problem****.*

This is precisely what we would like to do in reverse denotational semantics: obtain the *pointwise denotation* of a program, transform it into a subsidiary *pointfree denotation*, obtain the solution using the pointfree algebra of programs, and return back to the pointwise level where formal method practitioners are used to express their thoughts.

## 10 Algebra of programming

In a more respected than loved paper, John Backus [Bac78] 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. This line of thought has witnessed significant advances over the last decade, based on the so-called *functorial* approach to datatypes which, originating from [MA86], has reached the status of a thorough program calculus in [BdM97]. Because this style of calculation has become known as the *Bird-Meertens formalism* (BMF) [Bac88], we will refer to the transformation from the "$v$-space to the $p$-space" (where $v$ stands for *variable* and $p$ for *pointfree*) as the "BMF transformation".

## 11 Introducing the "BMF-transformation"

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 $loop\ g$ — merge 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

$$(f \cdot g)\ a = f(g\ a) \tag{9}$$

under typing rule

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

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:

$$\begin{array}{rcl} \langle f, g\rangle & : & A \longrightarrow B \times C \\ \langle f, g\rangle \; a & \stackrel{\mathrm{def}}{=} & \langle f \; a, g \; a\rangle \end{array} \tag{10}$$

*cf.* diagram

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

involving *projections* $\pi_1$ and $\pi_2$ which are such that $\pi_1\langle b, c\rangle = b$ and $\pi_2\langle b, c\rangle = c$. This is Backus [Bac78] *construction* operator, which can be specified as the following polymorphic operator in VDM-SL notation:

$$\begin{array}{l} \mathit{split}[@A, @B, @C] : (@A \to @B) \times (@A \to @C) \to @A \to (@B \times @C) \\ \mathit{split}\,(f, g)(a) \;\underline{\triangle} \\ \quad \textsf{mk-}\,(f\,(a), g\,(a)); \end{array} \tag{11}$$

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

$$\pi_1 \cdot \langle f, g\rangle = f \;,\; \pi_2 \cdot \langle f, g\rangle = g \tag{12}$$

are implicit in the diagram above, and the $\times$-*fusion law*

$$\langle g, h\rangle \cdot f = \langle g \cdot f, h \cdot f\rangle \tag{13}$$

expresses "left distribution" of composition of over *split*, a law in which two "parallel" *consumer* functions $g$ and $h$ fuse with another, *producer* function $f$.

Already known since Backus' *algebra of programs*, law (13) is an example of an equation "in the $p$-space". The compression of notation and the power of such laws is particularly apparent in dealing with recursion, that is, with inductive datatypes. For conciseness, we will focus on the datatype of *finite lists* (which is the one present in *word count*) and mention only the laws which are relevant for our calculations. (For a comprehensive account, see *eg.* [BdM97].)

Consider the following inductive definition (in VDM-SL) of the function which computes the sum of a list of integers:

$$\begin{array}{l} sum : \mathbb{Z}^* \to \mathbb{Z} \\ sum\,(l) \;\underline{\triangle} \\ \quad \textsf{if } l = [] \\ \quad \textsf{then } 0 \\ \quad \textsf{else hd } l + sum\,(\textsf{tl } l); \end{array} \tag{14}$$

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

$$
\begin{array}{l}
f\,(l) \triangleq \\
\quad \textsf{if } l = [] \\
\quad \textsf{then } k \\
\quad \textsf{else } g\,(\textsf{hd } l, f\,(\textsf{tl } l))
\end{array}
\tag{15}
$$

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 $\langle k, g\rangle$. Pairs of this kind are called *algebras* (= collections of functions and constants) which will be described in a compact way by resorting to a combinator which *dualizes* split (10):

$$
\begin{array}{rcl}
[f\,,g] & : & A + B \longrightarrow C \\
[f\,,g]\ x & \stackrel{\mathrm{def}}{=} & \left\{ \begin{array}{l} x = i_1\ a \Rightarrow f\ a \\ x = i_2\ b \Rightarrow g\ b \end{array} \right.
\end{array}
\tag{16}
$$

*cf.* diagram

$$
\begin{tikzcd}
A \arrow[r, "i_1"] \arrow[dr, "f"'] & A + B \arrow[d, "{[f\,,g]}"] & B \arrow[l, "i_2"'] \arrow[dl, "g"] \\
& C &
\end{tikzcd}
\tag{17}
$$

where $A + B$ denotes the disjoint union of $A$ and $B$.

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

$$
[\langle f\,, g\rangle, \langle h, k\rangle] = \langle [f\,, h], [g, k]\rangle
\tag{18}
$$

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

Thanks to the $[f\,, g]$ combinator, one can record the whole information about *algebra* $\langle k, g\rangle$ above into a single arrow

$$
B \xleftarrow{[\underline{k}, g]} 1 + A \times B
\tag{19}
$$

where 1 denotes an arbitrary (but fixed) singleton type — *eg.*

$$
Nil = \langle nil\rangle;
$$

in VDM-SL — and $\underline{k}$ denotes the "everywhere $k$" constant function such that

$$\underline{k} \cdot f = \underline{k} \tag{20}$$

Going further in the same direction, we can let arrow (19) participate in a larger diagram which records the whole "algorithmic" information about $f$ (15):

$$\begin{tikzcd} A^\star \arrow[d, "f"'] & 1 + A \times A^\star \arrow[l, "in"'] \arrow[d, "id+id\times f"] \\ B & 1 + A \times B \arrow[l, "{[\underline{k},g]}"] \end{tikzcd} \tag{21}$$

In this diagram: $A^\star$ is the VDM-SL type of finite lists (whose elements belong to $A$); $in$ is algebra $[\underline{[\,]}, cons]$ which builds $A^\star$-lists [5]; $id$ 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,

$$f \times g \stackrel{\mathrm{def}}{=} \langle f \cdot \pi_1, g \cdot \pi_2 \rangle \tag{22}$$

and its dual, the "sum" combinator:

$$f + g \stackrel{\mathrm{def}}{=} [i_1 \cdot f, i_2 \cdot g] \tag{23}$$

Diagram (21) expresses an equation about $f$

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

which we re-write into

$$f \cdot in = \alpha \cdot \mathsf{F}\, f \tag{24}$$

by introducing $\alpha = [\underline{k}, g]$ and $\mathsf{F}\, f = id + id \times f$.

Equation (24) is all we need to know to define $f$ — provided we instantiate $\alpha$, *ie.* $k$ and $g$. To express this uniqueness of $f$, dependent on $\alpha$, we write $(\!|\alpha|\!)$ — read "$\alpha$-catamorphism" — instead of $f$:

$$(\!|\alpha|\!) \cdot in = \alpha \cdot \mathsf{F}\, (\!|\alpha|\!) \tag{25}$$

For instance, catamorphism $(\!|[\underline{0}, (+)]|\!)$ is the "BMF-transformation" of the equivalent, four line VDM-SL (pointwise) definition of $sum$ (14). As an exercise, the reader can

[5] Operator $cons$ is defined in VDM-SL by

$$\begin{array}{l} cons[@A] : @A \times @A^* \to @A^* \\ cons\,(a, s)\ \underline{\triangle} \\ \quad [a] \curvearrowright s; \end{array}$$

recover this definition of $sum$ — or in general that of $f$ (15) — from (25) by applying standard laws of the algebra of programming known as $+$-*fusion*,

$$f \cdot [g, h] = [f \cdot g, f \cdot h] \tag{26}$$

and $+$-*absorption*

$$[g, h] \cdot (i + j) = [g \cdot i, h \cdot j] \tag{27}$$

among others.

Catamorphisms extend to any polynomial $\mathsf{F}$ and possess a number of remarkable properties, among which we select the *mutual-recursion law*, also called "Fokkinga law":

$$\begin{cases} f \cdot in = h \cdot \mathsf{F}\ \langle f, g\rangle \\ g \cdot in = k \cdot \mathsf{F}\ \langle f, g\rangle \end{cases} \equiv \langle f, g\rangle = (\!|\langle h, k\rangle|\!) \tag{28}$$

*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 is a formal basis for "parallel loop" inter-combination (or unravelling). It will play the major rôle in the calculations which remain to be done about $nw$.

## 12 Back to $nw$ — introduction of mutual recursion

Recall functions $nw$ and $nwrec$ in (8). We will now focus our attention on function $nwrec$ which, in the $p$-space,

$$\mathbb{Z} \times \mathbb{B} \xleftarrow{nwrec} \mathsf{char}^{\star}$$

is list catamorphism

$$(\!|[\langle \underline{0}, \underline{\mathsf{false}}\rangle, nwaux]|\!)$$

By delivering a pair of outputs, $nwaux$ (7) will split into

$$nwaux = \langle nwaux1, nwaux2\rangle$$

where

$$
\begin{aligned}
& nwaux1 : \mathsf{char} \times Acp \to \mathbb{Z} \\
& nwaux1\,(c, \mathsf{mk}\text{-}Acp\,(w, i)) \ \underline{\triangle} \\
& \quad \mathsf{if}\ sep\,(c) \vee i \\
& \quad \mathsf{then}\ w \\
& \quad \mathsf{else}\ w + 1; \\
& nwaux2 : \mathsf{char} \times Acp \to \mathbb{B} \\
& nwaux2\,(c, \text{-}) \ \underline{\triangle} \\
& \quad \neg\, sep\,(c);
\end{aligned}
$$

that is (in the $p$-space):

$$
\begin{aligned}
nwaux1 &= \vee \cdot (sep \times \pi_2) \to \pi_{12}, succ \cdot \pi_{12} \\
nwaux2 &= \neg \cdot sep \cdot \pi_1
\end{aligned}
$$

where $succ\ x = x + 1$, double projection $\pi_{ij}$ abbreviates $\pi_i \cdot \pi_j$, and $nwaux1$ resorts to *McCarthy's conditional* combinator defined by

$$
p \to g, h \stackrel{\text{def}}{=} [g, h] \cdot p? \tag{29}
$$

where

$$
(p?)a = \begin{cases} p\ \ a \ \Rightarrow i_1\ \ a \\ \neg(p\ \ a) \Rightarrow i_2\ \ a \end{cases} \tag{30}
$$

Function $nwrec$ can be reshaped

$$
\begin{aligned}
nwrec &= (\!|[\langle \underline{0}, \underline{\mathsf{false}} \rangle, \langle nwaux1, nwaux2 \rangle]|\!) \\
&= \quad \{ \text{exchange law (18)} \} \\
& \quad (\!|\langle [\underline{0}, nwaux1], [\underline{\mathsf{false}}, nwaux2] \rangle|\!)
\end{aligned}
$$

in a way so that it can be handled by the *mutual-recursion law* (28), for $h = [\underline{0}, nwaux1]$ and $k = [\underline{\mathsf{false}}, nwaux2]$. From this we infer $nwrec = \langle f, g \rangle$, as follows:

– Unknown $g$:

$$
\begin{aligned}
& g \cdot in = [\underline{\mathsf{false}}, nwaux2] \cdot \mathsf{F}\ \langle f, g \rangle \\
\equiv & \quad \{ \text{instantiations} \} \\
& g \cdot in = [\underline{\mathsf{false}}, \neg \cdot sep \cdot \pi_1] \cdot (id + id \times \langle f, g \rangle) \\
\equiv & \quad \{ +\text{-absorption (27)} \} \\
& g \cdot in = [\underline{\mathsf{false}}, \neg \cdot sep \cdot \pi_1 \cdot (id \times \langle f, g \rangle)] \\
\equiv & \quad \{ \times\text{-natural law (31) below} \} \\
& g \cdot in = [\underline{\mathsf{false}}, \neg \cdot sep \cdot \pi_1]
\end{aligned}
$$

One of the $\times$-*natural laws*

$$i \cdot \pi_1 = \pi_1 \cdot (i \times j) \tag{31}$$
$$j \cdot \pi_2 = \pi_2 \cdot (i \times j) \tag{32}$$

was used in this calculation. Back to the $v$-space, one obtains

$$\begin{cases} g\ [\,] = \mathsf{false} \\ g(cons(h,t)) = \neg(sep\ \ h) \end{cases}$$

– Unknown $f$: its diagram is

$$\begin{tikzcd} \mathsf{char}^* \arrow[d, "f"'] & 1 + \mathsf{char} \times \mathsf{char}^* \arrow[l, "in"'] \arrow[d, "id+id\times\langle f,g\rangle"] \\ \mathbb{Z} & 1 + \mathsf{char} \times (\mathbb{Z} \times \mathbb{B}) \arrow[l, "{[\underline{0},nwaux1]}"] \end{tikzcd}$$

Calculation:

$$\begin{array}{ll} & f \cdot in = [\underline{0}, nwaux1] \cdot (id + id \times \langle f, g\rangle) \\ \equiv & \{ \text{ instantiation of } nwaux \} \\ & f \cdot in = [\underline{0}, (\vee \cdot (sep \times \pi_2) \to \pi_{12}, succ \cdot \pi_{12})] \cdot (id + id \times \langle f, g\rangle) \\ \equiv & \{ \text{ +-absorption (27)} \} \\ & f \cdot in = [\underline{0}, (\vee \cdot (sep \times \pi_2) \to \pi_{12}, succ \cdot \pi_{12}) \cdot (id \times \langle f, g\rangle)] \\ \equiv & \{ \text{ McCarthy conditional and } \times\text{-cancellation (12)} \} \\ & f \cdot in = [\underline{0}, \vee \cdot (sep \times g) \to f \cdot \pi_2, succ \cdot f \cdot \pi_2] \end{array}$$

Back to the $v$-space:

$$\begin{cases} f\ [\,] = 0 \\ f(cons(h,t)) = \text{if } (sep\ \ h) \vee (g\ \ t) \\ \qquad\qquad\qquad\ \ \text{then } (f\ \ t) \\ \qquad\qquad\qquad\ \ \text{else } (f\ \ t) + 1 \end{cases}$$

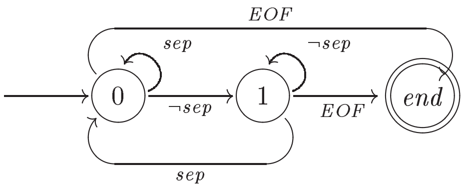
Finally,

$$\begin{array}{ll} & nw \\ = & \{ \text{ recall } nw(l) = nwrec(l).nw \} \\ & \pi_1 \cdot nwrec \\ = & \{ \text{ mutual recursion introduced above} \} \\ & \pi_1 \cdot \langle f, g\rangle \\ = & \{ \times\text{-cancellation (12)} \} \\ & f \end{array}$$

So we adopt $f$ as $nw$ in our final VDM-SL, which goes back into the $v$-space. For a more suggestive reading, we negate condition $sep\ h \vee g\ t$ and abstract $\neg g\ t$ into a "look ahead for word separator" predicate $sepahead$ [6]:

$nw$ : char$^*$ $\to \mathbb{Z}$
$nw\ (s) \triangleq$
  if $s = []$
  then $0$
  else if $\neg\ sep$ (hd $s$) $\wedge$ $sepahead$ (tl $s$)
    then $nw$ (tl $s$) $+ 1$
    else $nw$ (tl $s$);
$sepahead$ : char$^*$ $\to \mathbb{B}$
$sepahead\ (s) \triangleq$
  $(s = []) \vee sep$ (hd $s$)

This specification adds to our understanding of word counting mechanism: $nw$ counts the number of transitions from a non-separator to a separator character, or the end of the input stream. As anticipated earlier on, variable $inword$ has disappeared throughout this calculation. In fact, it can be regarded as a state "flag" implementing the "separator/non-separator" state automaton which is implicit in the original code:




Function $nw$ is now a proper inductive function: it belongs to the class of so-called *paramorphisms* [Mee92], which are very common in formal specification (for instance, the usual definition of the factorial function $n!$ is a paramorphism). Depicted in a diagram, $nw$ will look as follows

$$\begin{tikzcd} \text{char}^* \arrow[d, "nw"'] & 1 + \text{char} \times \text{char}^* \arrow[l, "{[\underline{[\,]}, cons]}"'] \arrow[d, "id + id \times \langle id, nw \rangle"] \\ \mathbb{Z} & 1 + \text{char} \times (\text{char}^* \times \mathbb{Z}) \arrow[l, "{[\underline{0}, h]}"] \end{tikzcd}$$

where

[6] The $\vee$ version of the if-then-else relies on VDM-SL's logic of partial functions [FL98].

$h : \mathsf{char} \times (\mathsf{char}^* \times \mathbb{Z}) \to \mathbb{Z}$
$h\,(c, \mathsf{mk\text{-}}\,(s, i)) \ \underline{\triangle}$
  $\mathsf{if}\ \neg\ sep\,(c) \wedge sepahead\,(s)$
  $\mathsf{then}\ i + 1$
  $\mathsf{else}\ i;$

All in all, we can write

$$[\![\ \texttt{main()}\ ]\!] = wc$$

where $wc = \langle nl, nw, \mathsf{len} \rangle$. In VDM-SL notation:

$wc : \mathsf{char}^* \to \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$
$wc\,(s) \ \underline{\triangle}$
  $\mathsf{mk\text{-}}\,(nl\,(s), nw\,(s), \mathsf{len}\ s);$

## 13 Summary

As pictured in Fig. 2, we have combined a formal method — *algebra of programming* [BdM97] — with a semi-formal one — *code slicing* [Wei81] — in order to perform the reverse specification of a little program already studied in the code slicing literature [GL91]: the *word count* program of [KR78]. We claim that we have gone deeper than [GL91] in understanding this piece of code.

A key point of the approach is its constructive style, based on a change of notation which leads to powerful algebraic laws of programming. This also helps in overcoming shortcomings of the specification language. For instance, the coproduct construct $A+B$ is not polymorphic in VDM-SL. For each particular $A$ and $B$, a specific disjoint union datatype has to be defined, *eg.* $AorB$ in:

$AorB = Left \mid Right;$
$Left :: value : A;$
$Right :: value : B;$

So injections $i_1$ and $i_2$ instantiate to $\mathsf{mk\text{-}}Left$ and $\mathsf{mk\text{-}}Right$, respectively, and $[f\,,g]$ to:

$either : (A \to C) \times (B \to C) \to AorB \to C$
$either\,(f, g)(x) \ \underline{\triangle}$
  $\mathsf{if\ is\text{-}}Left\,(x)$
  $\mathsf{then}\ f\,(x.value)$
  $\mathsf{else}\ g\,(x.value);$

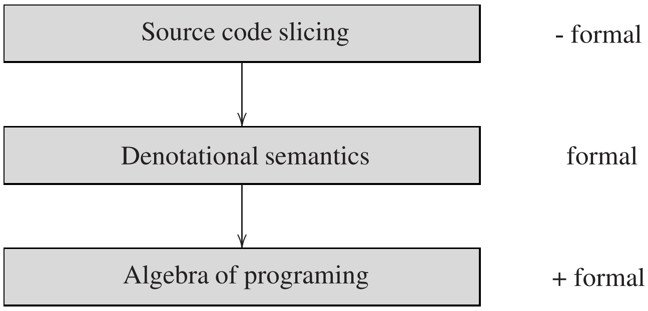



**Figure 2:** Formal/informal method combination.

Thanks to the change of notation, all our reasoning involving coproducts was handsomely carried out in the $p$-space (pointfree notation). Pointfree reasoning went as far as absorbing auxiliary variables and introducing mutual recursion, a specification mechanism which programmers never make explicit because they fear lacking efficiency.

Last but not least, the approach adds to program understanding in “cataloguing” semantic denotations into well-known classes of inductive schemata (cata/paramorphisms) which are rich in algebraic properties and are amenable to further reasoning (*eg.* re-implementation).

Slicing can be regarded as a denotational semantics “shortcut” — it trims down the complexity of handling all program variables at the same time. In fact, the *mutual-recursion law* (28) 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*,

$$\langle (\!| i |\!), (\!| j |\!) \rangle = (\!| (i \times j) \cdot \langle \mathsf{F}\ \pi_1, \mathsf{F}\ \pi_2 \rangle |\!) \tag{33}$$

which is a special case of (28).

## 14 Related work

Venkatesh [Ven91] was among the first to address program slicing from a formal semantics point of view. References [CLM95, CLM96] present approaches to recover (functional) specifications from imperative source code using “symbolic execution”. Specifications are expressed by pre/post-conditions. References [GC96, GC99] also base their reverse engineering strategies on pre-/post-conditions. Similarly, [CLM98] uses a first order logic formula to map a subset of the input program variables onto a set of initial states. This allows one to specify any initial state of the program. So, the slice calculated by adding such a condition on the input variable to the slicing criterion is called *conditioned slice*.

Reference [HDS95] proposes the use of program transformation techniques to construct a syntactically unrestricted slice, and so to improve the simplification power of slicing. In the same direction, [HD97] introduces the concept of *amorphous slicing*, which relaxes syntactic constraints traditional in slicing, thus generating smaller slices. In this way, the applicability of slicing to program comprehension is improved.

The repertoire of formal techniques for reverse engineering further includes "type inference" [vDM99] and concept/cluster analysis [vDK99]. These are applicable mainly to the detection of objects and can also be combined with techniques proposed in this paper. Likewise, semi-formal techniques [Vil01a] for the detection of recurrent algorithmic structures can also contribute to the identification — and later to the formalization — of program patterns.

## 15 Future work

The technique for code reversal 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. As anticipated in section 7 and discussed in [VO01], it needs further attention in presence of non-termination, forcing equality $=$ to give place to $\subseteq$ in (4) and thus a move towards the more general *relational algebra* of programming [Bac00, BdM97].

A forthcoming master thesis [Vil01b] is expected to present several exercises such as *word count* 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. Even within the functional interpretation, some program structures will require laws more powerful than those which we have been thinking of — for instance, comonadic calculations [Par00].

## Acknowledgements

The research described in this paper was carried out at the ALGORITMI R&D Center and was funded by the Portuguese Science and Technology Foundation (grant P060-P31B-09/97). The author wishes to thank Gustavo Villavicencio for his contribution to this paper, and Peter G. Larsen and Andreas Kerschbaumer for their comments. The use of VDMTools ® under a Free Academic Site License provided by IFAD is gratefully acknowledged.

## References

[Bac78] 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.

[Bac88] R. Backhouse. An exploration of the Bird-Meertens Formalism, July 1988. Department of Computing Science, Groningen University, The Netherlands.

[Bac00] 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.

[BdM97] R. Bird and O. de Moor. *Algebra of Programming.* Series in Computer Science. Prentice-Hall International, 1997. C. A. R. Hoare, series editor.

[BH93] R. C. Backhouse and P. F. Hoogendijk. Elements of a relational theory of datatypes. In S.A. Schuman, B. Möller, and H.A. Partsch, editors, *Formal Program Development*, number 755 in Lecture Notes in Computer Science, pages 7–42. Springer, 1993.

[CI90] E.J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[CLM95] Aniello Cimitile, Andrea De Lucia, and Malcolm C. Munro. Qualifying reusable functions using symbolic execution. In *Working Conference on Reverse Engineering, Toronto, Canada*, pages 178–187, 1995.

[CLM96] 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.

[CLM98] Aniello Cimitile, Andrea De Lucia, and Malcolm C. Munro. Conditioned program slicing. *Information and Software Technology (special issue on program slicing)*, 40(11/12):595–607, 1998.

[FL98] J. Fitzgerald and P.G. Larsen. *Modelling Systems: Practical Tools and Techniques for Software Development* . Cambridge University Press, 1st edition, 1998.

[GC96] 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.

[GC99] G.C. Gannod and B.H.C. Cheng. A formal apprach for reverse engineering: a case study. In *WCRE'99*, 1999.

[GL91] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[HD97] 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 Computer Society Press, Los Alamitos, California, USA, 1997.

[HDS95] 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.

[Jon86] C. B. Jones. *Systematic Software Development Using VDM*. Series in Computer Science. Prentice-Hall International, 1986. C. A. R. Hoare.

[KR78] B.W. Kernighan and D.M. Richtie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, N.J., 1978.

[Kre88] E. Kreyszig. *Advanced Engineering Mathematics*. John Wiley & Sons, Inc., 6th edition, 1988.

[MA86] E. G. Manes and M. A. Arbib. *Algebraic Approaches to Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1986. D. Gries, series editor.

[Mee92] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4:413–424, 1992.

[NSO99] 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.

[Par00] A. Pardo. Towards merging recursion and comonads. In *WGP'2000, Ponte de Lima*, July 2000.

[vDK99] Arie van Deursen and Tobias Kuipers. Identifying objects using cluster and concept analysis. *ICSE'99, ACM*, 1999.

[vDM99] Arie van Deursen and Leon Moonen. Understanding cobol systems using inferred types. *7th IWPC'99, IEEE Computer Society*, 1999.

[Ven91] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI)*, volume 26, pages 107–119, 1991.

[Vil01a] 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.

[Vil01b] G. Villavicencio. Formalización de una estrategia integral de ingeniería reversa, 2001. Master's thesis in preparation (University of San Luis, Argentina).

[VO01] G. Villavicencio and J.N. Oliveira. Formal reverse calculation supported by code slicing, 2001. To appear in *WCRE 2001: 8th Working Conference on Reverse Engineering* , 2-5 October 2001, Stuttgart, Germany.

[Wei81] Mark Weiser. Program slicing. In *Fifth International Conference on Software Engineering, San Diego, California*, March 1981.