Slicing Functional Programs by Calculation*

Nuno F. Rodrigues and Luís S. Barbosa
{nfr, lsb}@di.uminho.pt

Departamento de Informática, Universidade do Minho
4710-057 Braga, Portugal

Abstract. Program slicing is a well known family of techniques used to identify code fragments which depend on or are depended upon specific program entities. They are particularly useful in the areas of reverse engineering, program understanding, testing and software maintenance. Most slicing methods, usually targeting either the imperative or the object oriented paradigms, are based on some sort of graph structure representing program dependencies. Slicing techniques amount, therefore, to (sophisticated) graph transversal algorithms.
This paper proposes a completely different approach to the slicing problem for functional programs. Instead of extracting program information to build an underlying dependencies' structure, we resort to standard program calculation strategies, based on the so-called Bird-Meertens formalism. The slicing criterion is specified either as a projection or a hiding function which, once composed with the original program, leads to the identification of the intended slice. Going through a number of examples, the paper suggests this approach may be an interesting, even if not completely general alternative to slicing functional programs.

1 Introduction

By the end of the century *program understanding* emerged as a key concern in software engineering. 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 modify or improve running code. However, faced with so risky a dependence on legacy software, managers are more and more prepared to spend resources to increase confidence on — *i.e.*, the level of understanding of — their (otherwise untouchable) code. In fact the technological and economical relevance of *legacy* software as well as the complexity of their re-engineering entails the need for rigour. So it is likely that formal techniques developed for the production of fresh, high quality software will see the light of indutrial success in their reverse application to the analysis of running pre-existing code.

This paper focus on a particular program understanding technique — called *code slicing* [21, 19, 20] — which is reframed as a calculational problem in the *algebra of programming* [4]. More specifically, the process of computing program *slices*, *i.e.*, isolating parts of a program which depend on or are depended upon a specific computational entity, is reduced to the problem of solving an equation on the program denotational domain.

Program slicing, originally introduced in Wieser's thesis [19], is a family of techniques for restricting the behaviour of a program to some fragment of interest which, *e.g.*, contributes to the computation of a particular output or state variable. Slices are usually regarded as executable sub-programs extracted from source code by data and control flow analysis. Their computation is driven by what is referred to as a *slicing criterion*, which is, in most approaches, a pair containing a line number and a variable identifier. From the user point of view, this represents a point in the code whose impact she/he wants to inspect in the overall program. From the program slicer view, the slicing criterion is regarded as the *seed* from which a program slice is computed. According to Weiser original definition a slice consists of an executable sub-program including all statements with some direct or indirect consequence on the result of the value of the entity selected as the slicing criterion. The concern is to find only the pieces of code that affect a particular entity in the program. A basic distinction is drawn between *backwards* slicing which collects all data and

* The research reported in this paper is supported by FCT, under contract POSI/ICHS/44304/2002, in the context of the PURe project.

Dagstuhl Seminar Proceedings 05451
Beyond Program Slicing
http://drops.dagstuhl.de/opus/volltexte/2006/484

code fragments on which the slicing criterion depends, and *forward* slicing [9] which seeks for what depends on or is affected by it. A classical reference [6] show how the slices of a program ordered by set inclusion form a lattice where intersection corresponds to code sharing.

Slicing techniques are typically based on some form of abstract, graph-based representation of the program under scrutiny, from which dependence relations between the entities it manipulates can be identified and extracted. Therefore, in general, the slicing problem reduces to sub-graph identification with respect to a particular node. What kinds of computational entities can be represented in a node and what code dependencies does the underlying graph support are therefore the typical concerns.

As mentioned above, the approach sketched in this paper takes a completely different path. Instead of extracting program information to build an underlying dependencies' structure, we resort to standard program calculation strategies, based on the so-called Bird-Meertens formalism. The slicing criterion is specified either as a projection or a hiding function which, once composed with the original program, leads to the identification of the intended slice.

Slicing by calculation is, therefore, driven by the denotational semantics of the target program, as opposed to more classical syntax-oriented approaches documented in the literature (see *e.g.*, [] for an extended survey). To make calculation effective and concise we adopt the the *pointfree* style of expression [4] popularized among the functional programming community.

Finally, in order to keep presentation simple and avoid to distract the reader with denotational semantics technicalities, our approach is introduced in the context of the functional programming paradigm, in which program denotations are directly given in terms of functions between (partially ordered) sets [3]. The approach scales up, however, to imperative or object-oriented programs once the corresponding functional denotations have been computed.

The paper is organised as follows. Section 2 reviews some background concepts and results in program calculation within the Bird-Meertens formalism. Section 3 introduces the central intuitions of the proposed approach to slicing by calculation, distinguishing between *backward* and *forward* slicing. The next two sections illustrate both methods through examples. Finally section 6 concludes and points some directions for future work.

2 Algebra of Programming

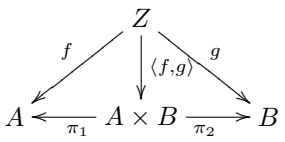
In his Turing Award lecture J. Backus [1] was among the first to advocate the need for programming languages which exhibit an *algebra* for reasoning about the objects it purport leading to the development of program calculi directly based on, actually driven by, type specifications. Since then this line of research has witnessed significant advances based on the *functorial* approach to datatypes [12] and reached the status of a program calculus in [4], building on top of a discipline of algorithm derivation and transformation which can be traced back to the so-called *Bird-Meertens formalism* [5, 11, 13] and the foundational work of T. Hagino [8].

In this paper we intend to build on this collection of *programming laws* to solve what will be referred in next section as *slicing equations*. Pointwise notation, as used in classical mathematics, involving operators as well as variable symbols, logical connectives, quantifiers, etc, is however inadequate to reason about programs in a concise and precise way. This justifies the introduction of a *pointfree* program denotation in which elements and function application are systematically replaced by functions and functional composition. The translation of the target program into an equivalent pointfree formulation is well studied in the program calculi community and shown to be made automatic to a large extent. In [14, 18] its role is compared to one played by the Laplace transform to solve differential equations in a linear space. This section provides a quick introduction to the pointfree algebra of programs.

Functional Glue. Recall that we have restricted our attention to functional programs, *i.e.*, pieces of code whose semantics can be expressed by functions f, g, h, ... Some functions have a particular role in the calculus: for example *identities* denoted by $\mathsf{id}_A : A \longleftarrow A$ or the so-called *final* functions $!_A : \mathbf{1} \longleftarrow A$ whose codomain is the singleton set denoted by $\mathbf{1}$ and consequently map every

element of A into the (unique) element of $\mathbf{1}$. Elements $x \in X$ are represented as *points*, *i.e.*, functions $\underline{x} : X \longleftarrow \mathbf{1}$, and therefore function application $f\,x$ can be expressed by composition $f \cdot \underline{x}$.

Functions can be *glued* in a number of ways which bare a direct correspondence with the ways programs may be assembled together. The most obvious one is *pipelining* which corresponds to standard functional composition denoted by $f \cdot g$ for $f : B \longleftarrow C$ and $g : B \longleftarrow A$. Functions with a common domain can be glued through a *split* $\langle f, g\rangle$ as shown in the following diagram:



which defines the product of two sets. Actually, the product of two sets A and B can be characterised either concretely (as the set of all pairs that can be formed by elements of A and B) or in terms of an abstract specification. In this case, we say set $A \times B$ is defined as the source of two functions $\pi_1 : A \longleftarrow A \times B$ and $\pi_2 : B \longleftarrow A \times B$, called the *projections*, which satisfy the following property: for any other set Z and arrows $f : A \longleftarrow Z$ and $g : B \longleftarrow Z$, there is a unique arrow $\langle f, g\rangle : A \times B \longleftarrow Z$, usually called the *split* of f and g, that makes the diagram above to commute. This can be said in a more concise way through the following equivalence which entails both an *existence* ($\Rightarrow$) and a *uniqueness* ($\Leftarrow$) assertion:

$$k = \langle f, g\rangle \quad \equiv \quad \pi_1 \cdot k = f \ \wedge \ \pi_2 \cdot k = g \tag{1}$$

Such an abstract characterization turns out to be more generic and suitable for conducting calculations. Let us illustrate this claim with a very simple example. Suppose we want to show that pairing projections of a cartesian product has no effect, *i.e.*, $\langle \pi_1, \pi_2\rangle = \mathsf{id}$. If we proceed in a concrete way we first attempt to convince ourselves that the unique possible definition for *split* is as a pairing function, *i.e.*, $\langle f, g\rangle\ z = \langle f\ z, g\ z\rangle$. Then, instantiating the definition for the case at hands, conclude

$$\langle \pi_1, \pi_2\rangle\ \langle x, y\rangle = \langle \pi_1\ \langle x, y\rangle, \pi_2\ \langle x, y\rangle\rangle = \langle x, y\rangle$$

Using the universal property (1) instead, the result follows immediately and in a *pointfree* way:

$$\mathsf{id} = \langle \pi_1, \pi_2\rangle \ \equiv \ \pi_1 \cdot \mathsf{id} = \pi_1 \wedge \pi_2 \cdot \mathsf{id} = \pi_2$$

Equation

$$\langle \pi_1, \pi_2\rangle \ = \ \mathsf{id}_{A\times B} \tag{2}$$

is called the *reflection* law for products. Similarly the following laws (known respectively as $\times$ *cancelation*, *fusion* and *absorption*) are derivable from (1):

$$\pi_1 \cdot \langle f, g\rangle = f \ , \pi_2 \cdot \langle f, g\rangle = g \tag{3}$$

$$\langle g, h\rangle \cdot f \ = \ \langle g \cdot f, h \cdot f\rangle \tag{4}$$

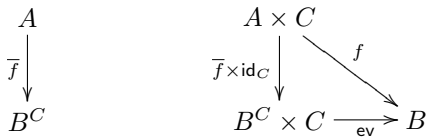
$$(i \times j) \cdot \langle g, h\rangle \ = \ \langle i \cdot g, j \cdot h\rangle \tag{5}$$

The same applies to *structural equality*:

$$\langle f, g\rangle = \langle k, h\rangle \ \equiv \ f = k \ \wedge \ g = h \tag{6}$$

Finally note that the product construction applies not only to sets but also to functions, yielding, for $f : B \longleftarrow A$ and $g : B' \longleftarrow A'$, function $f \times g : B \times B' \longleftarrow A \times A'$ defined as the split $\langle f \cdot \pi_1, g \cdot \pi_2\rangle$. This equivales to the following pointwise definition: $f \times g = \lambda\,\langle a, b\rangle\ .\ \langle f\ a, g\ b\rangle$.

Notation B^A is used to denote *function space*, *i.e.*, the set of (total) functions from A to B. It is also characterised by an universal property: for all function $f : B \longleftarrow A \times C$, there exists a unique $\overline{f} : B^C \longleftarrow A$, called the *curry* of f, such that $f = \mathsf{ev} \cdot (\overline{f} \times C)$. Diagrammatically,

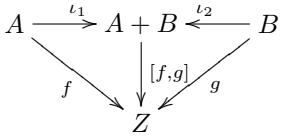


i.e.,

$$k = \overline{f} \quad \equiv \quad f = \mathsf{ev} \cdot (k \times \mathsf{id}) \tag{7}$$

Dually, functions sharing the same codomain may be glued together through an *either* combinator, expressing alternative behaviours, and introduced as the universal arrow in a datatype sum construction.

The *sum* $A+B$ (or *coproduct*) of A and B corresponds to their disjoint union. The construction is dual to the product one. From a programming point of view it corresponds to the aggregation of two entities in *time* (as in a `union` construction in C), whereas product entails an aggregation in *space* (as a `record`). It also arises by universality: $A + B$ is defined as the target of two arrows $\iota_1 : A + B \longleftarrow A$ and $\iota_2 : A + B \longleftarrow B$, called the *injections*, which satisfy the following universal property: for any other set Z and functions $f : Z \longleftarrow A$ and $g : Z \longleftarrow B$, there is a unique arrow $[f, g] : Z \longleftarrow A + B$, usually called the *either* (or *case*) of f and g, that makes the following diagram to commute:



Again this universal property can be written as

$$k = [f, g] \quad \equiv \quad k \cdot \iota_1 = f \ \wedge \ k \cdot \iota_2 = g \tag{8}$$

from which one infers correspondent *cancelation*, *reflection* and *fusion* results:

$$[f, g] \cdot \iota_1 = f \ , \ [f, g] \cdot \iota_2 = g \tag{9}$$

$$[\iota_1, \iota_2] \ = \ \mathsf{id}_{X+Y} \tag{10}$$

$$f \cdot [g, h] \ = \ [f \cdot g, f \cdot h] \tag{11}$$

Products and sums interact through the following *exchange* law

$$[\langle f, g\rangle, \langle f', g'\rangle] \ = \ \langle [f, f'], [g, g']\rangle \tag{12}$$

provable by either product (1) or sum (8) universality. The *sum* combinator also applies to functions yielding $f + g : A' + B' \longleftarrow A + B$ defined as $[\iota_1 \cdot f, \iota_2 \cdot g]$.

Conditional expressions are modelled by coproducts. In this paper we adopt the McCarthy conditional constructor written as $(p \ \rightarrow \ f, \ g)$, where $p : \mathbf{2} \longleftarrow A$ is a predicate. Intuitively, $(p \ \rightarrow \ f, \ g)$ reduces to f if p evaluates to true and to g otherwise. The conditional construct is defined as

$$(p \ \rightarrow \ f, \ g) \ = \ \langle f, g\rangle \cdot p?$$

where $p? : A + A \longleftarrow A$ is determined by predicate p as follows

$$p? = \qquad A \xrightarrow{[\mathsf{id},p]} A \times (\mathbf{1} + \mathbf{1}) \xrightarrow{\mathsf{dl}} A \times \mathbf{1} + A \times \mathbf{1} \xrightarrow{\pi_1+\pi_1} A + A$$

where dl is the distributivity isomorphism. The following laws are usefull to calculate with conditionals [7].

$$h \cdot (p \rightarrow f, g) = (p \rightarrow h \cdot f, h \cdot g) \quad (13)$$

$$(p \rightarrow f, g) \cdot h = (p \cdot h \rightarrow f \cdot h, g \cdot h) \quad (14)$$

$$(p \rightarrow f, g) = (p \rightarrow (p \rightarrow f, g), (p \rightarrow f, g)) \quad (15)$$

Recursion. Recursive functions over inductive datatypes (such as finite sequences or binary trees) are given by their *genetic* information, *i.e.*, the specification of what is to be done in an instance of a recursive call. Consider, for example, the pointfree specification of the function which computes the length of a list $\mathsf{len} : \mathbb{N} \longleftarrow A^*$. A^* is an example of an inductive type: its elements are built by one of the following *constructors*: $\mathsf{nil} : A^* \longleftarrow \mathbf{1}$, which builds the empty list, and $\mathsf{cons} : A^* \longleftarrow A \times A^*$, which appends an element to the head of the list. The two constructors are glued by an *either* $\mathsf{in} = [\mathsf{nil}, \mathsf{cons}]$ whose codomain is an instance of polynomial functor $\mathsf{F}\, X = \mathbf{1} + A \times X$. The algorithm contents of function len is exposed in the following diagram:

$$\begin{array}{ccc} \mathbf{1} + A \times \mathbb{N} & \xrightarrow{[\underline{0}, \mathsf{succ} \cdot \pi_2]} & \mathbb{N} \\ \uparrow{\scriptstyle \mathsf{id}+\mathsf{id}\times\mathsf{len}} & & \uparrow{\scriptstyle \mathsf{len}} \\ \mathbf{1} + A \times A^* & \xrightarrow{\mathsf{in}=[\mathsf{nil},\mathsf{cons}]} & A^* \end{array}$$

where the 'genetic' information is given by $[\underline{0}, \mathsf{succ} \cdot \pi_2]$: either return 0 or the successor of the value computed so far. Function len, being entirely determined by its 'gene' is said its *inductive extension* or *catamorphism* and represented by $(\!|[\underline{0}, \mathsf{succ} \cdot \pi_2]|\!)$.

Catamorphisms extend to any polynomial F and possess a number of remarkable properties, *e.g.*,

$$(\!|in|\!) = \mathsf{id} \quad (16)$$

$$(\!|g|\!) \cdot \mathsf{in} = g \cdot \mathsf{F}\,(\!|g|\!) \quad (17)$$

$$f \cdot (\!|g|\!) = (\!|h|\!) \Leftarrow f \cdot g = h \cdot \mathsf{F}\, f \quad (18)$$

$$(\!|g|\!) \cdot \mathsf{T}\, f = (\!|g \cdot \mathsf{F}\,(f, \mathsf{id})|\!) \quad (19)$$

where T is the functor that assigns to a set X the corresponding inductive type for F (in our example, $\mathsf{T}\, X = X^*$). Laws above are called, respectively, cata-reflection, -cancelation, -fusion and -absorption.

3 Slicing by Calculation

As mentioned in section 1, mainstream research on program slicing targets imperative languages and, therefore, is oriented towards particular, well characterised notions of computational variable, program statement and control flow behaviour. Slicing algorithms exploit graph representations of the target program reflecting dependencies between instances of such notions.

Slicing functional programs requires a different perspective. Functions, rather than program statements, are the basic computational units and functional composition replaces statement sequencing. Moreover there is no notion of assignable variable or global state whatsoever.

This section introduces the basic strategy of our approach to slicing by calculation. The idea is to resort to special functions (called either *projections* or *hiding* functions in the sequel) to slice other functions. Let f be the target functional program. Typically f receives a number of arguments and returns a number of results packed together in a multiplicative context which may, eventually, be embedded in contexts represented by functors R and T respectively. Formally,

$$f : \mathsf{T}(\Pi_{j \in J}\, B_j) \longleftarrow \mathsf{R}(\Pi_{i \in I}\, A_i) \quad (20)$$

A *projection* function $\pi : \mathsf{T}(B_k) \longleftarrow \mathsf{T}(\Pi_{j \in J}\, B_j)$ selects a particular factor in the output of f. The effect of its composition with f is to find out the trace of such a factor on the body of f. As it proceeds backwards, we call its computation a *backward* slicing process, in a way which is consistent with the corresponding designation in conventional slicing. Formally the backward slicing problem is stated in terms of solving for unknown f' the following equation:

$$\pi \cdot f \;=\; f' \tag{21}$$

Note that this approach is not based in any intermediate program representation (like *Control Flow Graphs* or *Data Dependence Graphs*), the projection function itself acting as the *slicing criterion*. The slicing problem reads: find a new function f' corresponding to the restriction of f with respect to π. At first it might seem complex to come up with a suitable projection function to encode a particular restriction (and compute the corresponding slice). In most cases, however, as illustrated in the examples discussed in the following section, a simple product projection, like a selector in a pair or a datatype destructor, is enough.

The dual process corresponds to what is traditionally called *forward* slicing. In this case a function $\sigma : \mathsf{R}(\Pi_{i \in I}\, A_j) \longleftarrow \mathsf{R}(\Pi_{i \in I}\, A_j)$ is used to *hide* from the input of f the arguments one does *not* want to consider as a slicing criteria. Hiding a particular factor A_k in a product $\Pi_{i \in I}\, A_i$ amounts to apply to that factor function $\bot : A_k \longleftarrow A_k$ which maps any value of type A_k to the bottom element $\bot_{A_k}$ of the corresponding data domain[1]. Formally,

$$\bot \;=\; A_k \xrightarrow{\;!\;} \mathbf{1} \xrightarrow{\;\bot_{A_k}\;} A_k \tag{22}$$

Therefore, hiding function σ takes the form of a product of $\bot$ functions and identities embedded in context R. The *forward* slicing problem is then stated in terms of solving for unknown f' the following equation:

$$f \cdot \sigma \;=\; f' \tag{23}$$

The following two sections illustrate the application of this method to a few small examples.

4 Functional Backward Slicing

This section introduces two examples of backward slicing by calculation. In the first case, the well-known word count `wc` example, the output of the target program is a product of two naturals corresponding to the number of lines and characters in an input string. The projection function is therefore a single product projection. The second example is a toy bank account management system whose output is a user defined datatype.

In both cases we start with a translation of the original program f into a pointfree notation. Once chosen a suitable slicing criterion π, the slicing process proceeds by calculation of $\pi \cdot f$ resorting to the Bird-Mertens formalism and calculus. Then the computed slice can be used directly (if the programming language allows the direct codification of pointfree expressions, which is the case of *e.g.*, HASKELL) or first translated back to a more conventional pointwise notation.

4.1 Slicing With Tuples

Consider the problem of identifying a *slice* in the following functional version of the Unix word-count utility (`wc`), with the `-lc` flag.

```
wc = wcAux (1,0)

wcAux :: (Int, Int) -> String -> (Int, Int)
```

[1] Recall that in the semantics of functional programs data domains are modelled by flat partial orders with a bottom element representing undefinedness.

```
wcAux p []           = p
wcAux (lc, cc) (h:t) =
    if h == '\n' then wcAux (lc+1, cc+1) t
                 else wcAux (lc, cc+1) t
```

which is translated into the Bird-Merteens formalism as

$$(\!|[\langle \underline{1}, \underline{0}\rangle, [(succ \times succ) \cdot \pi_2, (id \times succ) \cdot \pi_2] \cdot p?]|\!)_{\mathsf{F}}$$

where $p = ((' \backslash n' ==) \cdot \pi_1)?$.

The original function counts the number of lines and characters for a given text input. Our goal is to identify a slice of `wc` which just computes the number of lines. This is given by the first component of the pair returned by the original `wc` program. Thus, it is expectable that a function that selects the first element of a pair constitutes a good candidate for a slicing criterion. Indeed we shall use $\pi_1 =$ `fst :: (a, b) -> a`, the HASKELL first projection of a pair. Thus the slicing problem reduces to solving the following equation:

$$f' \;=\; \pi_1 \cdot (\!|[\langle \underline{1}, \underline{0}\rangle, [(succ \times succ) \cdot \pi_2, (id \times succ) \cdot \pi_2] \cdot p?]|\!)_{\mathsf{F}}$$

which leads to the following calculation

$$\begin{array}{ll}
& f' = \qquad \{f' \text{ definition} \\
& \pi_1 \cdot (\!|[\langle \underline{1}, \underline{0}\rangle, [(succ \times succ) \cdot \pi_2, (id \times succ) \cdot \pi_2] \cdot p?]|\!)_{\mathsf{F}} \\
= & \quad \{\text{composing with } \mathsf{id}\ \} \\
& \pi_1 \cdot (\!|[\langle \underline{1}, \underline{0}\rangle, [((succ \times succ) \cdot id) \cdot \pi_2, ((id \times succ) \cdot id) \cdot \pi_2] \cdot p?]|\!)_{\mathsf{F}} \\
= & \quad \{\text{reflection-}\times \text{ (twice)}\} \\
& \pi_1 \cdot (\!|[\langle \underline{1}, \underline{0}\rangle, [((succ \times succ) \cdot \langle \pi_1, \pi_2\rangle) \cdot \pi_2, ((id \times succ) \cdot \langle \pi_1, \pi_2\rangle) \cdot \pi_2] \cdot p?]|\!)_{\mathsf{F}} \\
= & \quad \{\text{absorption-}\times \text{ (twice)}\} \\
& \pi_1 \cdot (\!|[\langle \underline{1}, \underline{0}\rangle, [\langle succ \cdot \pi_1, succ \cdot \pi_2\rangle \cdot \pi_2, \langle id \cdot \pi_1, succ \cdot \pi_2\rangle \cdot \pi_2] \cdot p?]|\!)_{\mathsf{F}} \\
= & \quad \{\text{fusion-}\times \text{ (twice)}\} \\
& \pi_1 \cdot (\!|[\langle \underline{1}, \underline{0}\rangle, [\langle (succ \cdot \pi_1) \cdot \pi_2, (succ \cdot \pi_2) \cdot \pi_2\rangle, \langle (id \cdot \pi_1) \cdot \pi_2, (succ \cdot \pi_2) \cdot \pi_2\rangle] \cdot p?]|\!)_{\mathsf{F}} \\
= & \quad \{\text{exchange law, identities }\} \\
& \pi_1 \cdot (\!|[\langle \underline{1}, \underline{0}\rangle, \langle [(succ \cdot \pi_1) \cdot \pi_2, \pi_1 \cdot \pi_2], [(succ \cdot \pi_2) \cdot \pi_2, (succ \cdot \pi_2) \cdot \pi_2]\rangle \cdot p?]|\!)_{\mathsf{F}} \\
= & \quad \{\text{fusion-}\times\} \\
& \pi_1 \cdot (\!|[\langle \underline{1}, \underline{0}\rangle, \langle [(succ \cdot \pi_1) \cdot \pi_2, \pi_1 \cdot \pi_2] \cdot p?, [(succ \cdot \pi_2) \cdot \pi_2, (succ \cdot \pi_2) \cdot \pi_2] \cdot p?\rangle]|\!)_{\mathsf{F}} \\
= & \quad \{[f, f] \cdot p? = f\} \\
& \pi_1 \cdot (\!|[\langle \underline{1}, \underline{0}\rangle, \langle [(succ \cdot \pi_1) \cdot \pi_2, \pi_1 \cdot \pi_2] \cdot p?, (succ \cdot \pi_2) \cdot \pi_2\rangle]|\!)_{\mathsf{F}} \\
= & \quad \{\text{exchange law}\} \\
& \pi_1 \cdot (\!|\langle [\underline{1}, [(succ \cdot \pi_1) \cdot \pi_2, \pi_1 \cdot \pi_2] \cdot p?], [\underline{0}, (succ \cdot \pi_2) \cdot \pi_2]\rangle|\!)_{\mathsf{F}} \\
= & \quad \{\text{absorption-}\times \text{ (twice), cancelation-}\times \text{ (twice)}\} \\
& \pi_1 \cdot (\!|\langle [\underline{1}, [(succ \cdot \pi_2) \cdot (id \times \pi_1) \cdot \langle \pi_1, \pi_2\rangle, \pi_2 \cdot (id \times \pi_1) \cdot \langle \pi_1, \pi_2\rangle] \cdot p?], [\underline{0}, (succ \cdot \pi_2) \cdot \pi_2]\rangle|\!)_{\mathsf{F}} \\
= & \quad \{\text{reflexion-}\times \text{ (twice), fusion-}+\text{, definition of } p\} \\
& \pi_1 \cdot (\!|\langle [\underline{1}, [succ \cdot \pi_2, \pi_2] \cdot (id \times \pi_1) + (id \times \pi_1) \cdot ((' \backslash n' ==) \cdot \pi_1)?], [\underline{0}, (succ \cdot \pi_2) \cdot \pi_2]\rangle|\!)_{\mathsf{F}} \\
= & \quad \{\text{absorption-}\times\text{, cancelation-}\times\text{, reflexion-}\times\} \\
& \pi_1 \cdot (\!|\langle [\underline{1}, [succ \cdot \pi_2, \pi_2] \cdot (id \times \pi_1) + (id \times \pi_1) \cdot ((' \backslash n' ==) \cdot \pi_1 \cdot (id \times \pi_1))?], [\underline{0}, (succ \cdot \pi_2) \cdot \pi_2]\rangle|\!)_{\mathsf{F}} \\
= & \quad \{\text{McCarthy conditional fusion law}\}
\end{array}$$

$$
\begin{aligned}
& \pi_1 \cdot (\!|\langle [\underline{1}, [succ \cdot \pi_2, \pi_2] \cdot p? \cdot (id \times \pi_1)], [\underline{0}, (succ \cdot \pi_2) \cdot \pi_2] \rangle |\!)_{\mathsf{F}} \\
= & \quad \{\text{natural-}id\text{, absorption-}+\text{, absorption-}\times\text{, cancelation-}\times\text{, reflexion-}\times\} \\
& \pi_1 \cdot (\!|\langle [\underline{1}, [succ \cdot \pi_2, \pi_2] \cdot p?] \cdot id + (id \times \pi_1), [\underline{0}, (succ \cdot \pi_2)] \cdot id + (id \times \pi_2) \rangle |\!)_{\mathsf{F}} \\
= & \quad \{\text{definition of Functor } \mathsf{F}\} \\
& \pi_1 \cdot (\!|\langle [\underline{1}, [succ \cdot \pi_2, \pi_2] \cdot p?] \cdot \mathsf{F}\ \pi_1, [\underline{0}, (succ \cdot \pi_2)] \cdot \mathsf{F}\ \pi_2 \rangle |\!)_{\mathsf{F}} \\
= & \quad \{\text{corollary of the Fokkinga law [4]}\} \\
& \pi_1 \cdot \langle (\!| [\underline{1}, [succ \cdot \pi_2, \pi_2] \cdot p?] |\!)_{\mathsf{F}}, (\!| [\underline{0}, succ \cdot \pi_2] |\!)_{\mathsf{F}} \rangle \\
= & \quad \{\text{cancelation-}\times\} \\
& (\!| [\underline{1}, [succ \cdot \pi_2, \pi_2] \cdot p?] |\!)_{\mathsf{F}}
\end{aligned}
$$

Finally the result can be translated back to concrete HASKELL syntax, yielding the following program:

```
wc = foldr
      (\c -> if c == '\n' then succ else id)
      1
```

or, going pointwise,

```
wc = wcAux 1

wcAux :: Int -> String -> Int
wcAux p []     = p
wcAux lc (h:t) =
    if h == '\n' then wcAux lc+1 t
                 else wcAux lc t
```

A similar approach could be taken, using π_2 as a slicing criterion, to isolate the character count computation inside `wc`.

4.2 Slicing With User Defined Data Types

Our second example is a migration function which feeds a new model of a (toy) bank account system with data coming form an hypothetic legacy system. The new model records information on account balances and holders in structured according to the following HASKELL datatype declaration:

```
data System = Sys { clients  :: [Client],
                    accounts :: [Account] }

data Client = Clt { cltid :: CltId,
                    name  :: CltName }

data Account = Acc { accid  :: AccId,
                     holder :: CltId,
                     amount :: Amount }
```

Note that datatype `System` has `Sys :: [Client] -> [Account] -> System` as its constructor, and `clients :: System -> [Client]` and `accounts :: System -> [Account]` as destructors, retrieving the list of clients and accounts, respectively. Also notice that in HASKELL type constructors like `System` are curried — a detail one has to keep in mind along the slice calculation. Datatypes `Client` and `Account` are defined similarly.

The target program is now function `migrate` which accepts a sequence of tuples, containing information about bank clients and their respective balances and populates datatype `System`. The function procedes recursively on the input sequence and is therefore given as a *catamorphism* for a particular 'gene'. Consider the following HASKELL definition of `migrate` where the underlying catamorphism is encoded in the standard `foldr` primitive function.

```
migrate :: [((CltId, CltName), (AccId, Amount))] -> System
migrate =
  foldr (curry $ (uncurry Sys) . split f1 f2)
        (Sys [] [])
  where f1 = uncurry (:) . split (uncurry Clt . p1 . p1)
                                 (clients . p2)
        f2 = uncurry (:) . split ((uncurry . uncurry $ Acc) . split (split (p1 . p2 . p1)
                                                                           (p1 . p1 . p1))
                                                                   (p2 . p2 . p1))
                                 (accounts . p2)
```

A direct translation of this program to the Bird-Meertens formalism leads to

$$\begin{aligned}
\texttt{migrate} &= (\!|[\underline{Sys\ \mathsf{nil}\ \mathsf{nil}}, \overline{Sys} \cdot \langle f_1, f_2 \rangle]|)_{\mathsf{F}} \\
f_1 &= \mathsf{cons} \cdot \langle \overline{Clt} \cdot \pi_1 \cdot \pi_1, clients \cdot \pi_2 \rangle \\
f_2 &= \mathsf{cons} \cdot \langle \overline{\overline{Acc}} \cdot \langle \langle \pi_1 \cdot \pi_2 \cdot \pi_1, \pi_1 \cdot \pi_1 \cdot \pi_1 \rangle, \pi_2 \cdot \pi_2 \cdot \pi_1 \rangle, accounts \cdot \pi_2 \rangle
\end{aligned}$$

The problem now is to compute the two slices of `migrate` which correspond to the information on clients and accounts so that client and account migration can be done by two independent processes. To identify the clients slice we take the original `clients` selector as the slicing criteria. The slice itself will be again a recursive function and therefore it is written as catamorphism $(\!|g|\!)$ over sequences. The equation we are left to solve is

$$(\!|g|\!) = clients \cdot (\!|[\underline{Sys\ \mathsf{nil}\ \mathsf{nil}}, \overline{Sys} \cdot \langle f_1, f_2 \rangle]|\!) \tag{24}$$

Following by calculation over the f value, one as

$$\begin{aligned}
& (\!|g|\!) = clients \cdot (\!|[\underline{Sys\ \mathsf{nil}\ \mathsf{nil}}, \overline{Sys} \cdot \langle f_1, f_2 \rangle]|\!) \\
\Leftarrow \quad & \{\text{cata-fusion }\} \\
& g \cdot (\mathsf{F}\ clients) = clients \cdot [\underline{Sys\ \mathsf{nil}\ \mathsf{nil}}, \overline{Sys} \cdot \langle f_1, f_2 \rangle] \\
\Leftrightarrow \quad & \{\text{ constant function and uncurry }\} \\
& g \cdot (\mathsf{F}\ clients) = clients \cdot [\overline{Sys} \cdot \langle \underline{\mathsf{nil}}, \underline{\mathsf{nil}} \rangle, \overline{Sys} \cdot \langle f_1, f_2 \rangle] \\
\Leftrightarrow \quad & \{\text{absorption-}+\} \\
& g \cdot (\mathsf{F}\ clients) = [clients \cdot (\overline{Sys} \cdot \langle \underline{\mathsf{nil}}, \underline{\mathsf{nil}} \rangle), clients \cdot \overline{Sys} \cdot \langle f_1, f_2 \rangle] \\
\Leftrightarrow \quad & \{\text{definition of } clients, \cdot \text{ is associative}\} \\
& g \cdot (\mathsf{F}\ clients) = [\pi_1 \cdot \langle \underline{\mathsf{nil}}, \underline{\mathsf{nil}} \rangle, \pi_1 \cdot \langle f_1, f_2 \rangle] \\
\Leftrightarrow \quad & \{\text{cancelation-}\times \text{ (twice)}\} \\
& g \cdot (\mathsf{F}\ clients) = [\underline{\mathsf{nil}}, f_1] \\
\Leftrightarrow \quad & \{\text{definition of } f_1\} \\
& g \cdot (\mathsf{F}\ clients) = [\underline{\mathsf{nil}}, \mathsf{cons} \cdot \langle \overline{Clt} \cdot \pi_1 \cdot \pi_1, clients \cdot \pi_2 \rangle] \\
\Leftrightarrow \quad & \{\text{cancelation-}\times \text{ (twice)}\} \\
& g \cdot (\mathsf{F}\ clients) = [\underline{\mathsf{nil}}, \mathsf{cons} \cdot \langle \overline{Clt} \cdot \pi_1 \cdot \pi_1 \cdot \langle \pi_1, clients \cdot \pi_2 \rangle, \pi_2 \cdot \langle \pi_1, clients \cdot \pi_2 \rangle \rangle] \\
\Leftrightarrow \quad & \{\text{absorption-}\times \text{ (twice)}\} \\
& g \cdot (\mathsf{F}\ clients) = [\underline{\mathsf{nil}}, \mathsf{cons} \cdot \langle \overline{Clt} \cdot \pi_1 \cdot \pi_1 \cdot (id \times clients) \cdot \langle \pi_1, \pi_2 \rangle, \pi_2 \cdot (id \times clients) \cdot \langle \pi_1, \pi_2 \rangle \rangle] \\
\Leftrightarrow \quad & \{\text{natural-}id \text{ (twice), reflection-}\times \text{ (twice)}\} \\
& g \cdot (\mathsf{F}\ clients) = [\underline{\mathsf{nil}}, \mathsf{cons} \cdot \langle \overline{Clt} \cdot \pi_1 \cdot \pi_1 \cdot (id \times clients), \rangle] \\
\Leftrightarrow \quad & \{\text{fusion-}\times, \text{ def. of functor } \mathsf{F}\}
\end{aligned}$$

$$
\begin{array}{ll}
& g \cdot (id + id \times clients) = [\underline{\mathsf{nil}}, \mathsf{cons} \cdot \langle \overline{Clt} \cdot \pi_1 \cdot \pi_1, \pi_2 \rangle] \cdot (id \times clients) \\
\Leftarrow & \quad \{\text{function equality }\} \\
& g = [\underline{\mathsf{nil}}, \mathsf{cons} \cdot \langle \overline{Clt} \cdot \pi_1 \cdot \pi_1, \pi_2 \rangle]
\end{array}
$$

The computed slice is then

$$(\!|[\underline{\mathsf{nil}}, \mathsf{cons} \cdot \langle \overline{Clt} \cdot \pi_1 \cdot \pi_1, \pi_2 \rangle]|\!)$$

which can be translated to HASKELL in a straghtforward way as

```
clientMigration :: [((CltId, CltName), (AccId, Amount))] -> [Client]
clientMigration = foldr (curry $ (uncurry (:) . split (uncurry Clt . p1 . p1) p2)) []
```

A similar calculation isolates the account information migration process. Note that a reverse calculation starting with the two slices would lead again to the original function, providing evidence of the semantic soundness of this approach. Such is difficult to prove in conventional slicing techniques giving their essentially *syntactic* nature.

5 Functional Forward Slicing

To illustrate forward slicing calculation we resort again to the toy bank system and function `migration` defined in the previous section. The idea is to hide client information from the input in order to compute the forward slice relative to account information. By type inspection it comes clear that the relevant hiding function in this case is

$$\sigma \ = \ (\bot \times \mathsf{id})^* \tag{25}$$

Then,

$$
\begin{array}{ll}
& f' = (\!|[\underline{Sys \ \mathsf{nil} \ \mathsf{nil}}, \overline{Sys} \cdot \langle f_1, f_2 \rangle]|\!) \cdot (\bot \times id)^* \\
\Leftrightarrow & \quad \{\text{cata-absorption}\} \\
& f' = (\!|[\underline{Sys \ \mathsf{nil} \ \mathsf{nil}}, \overline{Sys} \cdot \langle f_1, f_2 \rangle] \cdot (id + (\bot \times id) \times id)|\!) \\
\Leftrightarrow & \quad \{\text{absorption-+, identities}\} \\
& f' = (\!|[\underline{Sys \ \mathsf{nil} \ \mathsf{nil}}, \overline{Sys} \cdot \langle f_1, f_2 \rangle \cdot (\bot \times id) \times id]|\!) \\
\Leftrightarrow & \quad \{\text{fusion-}\times \text{ (twice)}\} \\
& f' = (\!|[\underline{Sys \ \mathsf{nil} \ \mathsf{nil}}, \overline{Sys} \cdot \langle f_1 \cdot (\bot \times id) \times id, f_2 \cdot (\bot \times id) \times id \rangle]|\!)
\end{array}
$$

Let us now compute separately expressions $f_1 \cdot ((\bot \times id) \times id)$ and $f_2 \cdot ((\bot \times id) \times id)$:

$$
\begin{array}{ll}
& f_1 \cdot ((\bot \times id) \times id) \\
\Leftrightarrow & \quad \{\text{definition of } f_1\} \\
& \mathsf{cons} \cdot \langle \overline{Clt} \cdot \pi_1 \cdot \pi_1, clients \cdot \pi_2 \rangle \cdot ((\bot \times id) \times id) \\
\Leftrightarrow & \quad \{\text{fusion-}\times\} \\
& \mathsf{cons} \cdot \langle \overline{Clt} \cdot \pi_1 \cdot \pi_1 \cdot ((\bot \times id) \times id), clients \cdot \pi_2 \cdot ((\bot \times id) \times id) \rangle \\
\Leftrightarrow & \quad \{\text{corolary of the cancelation law for } \times\} \\
& \mathsf{cons} \cdot \langle \overline{Clt} \cdot \pi_1 \cdot (\bot \times id) \cdot \pi_1, clients \cdot id \cdot \pi_2 \rangle \\
\Leftrightarrow & \quad \{\text{corolary of the cancelation law for } \times, \text{ identity and } ! \ \} \\
& \mathsf{cons} \cdot \langle \overline{Clt} \cdot \bot, clients \cdot \pi_2 \rangle
\end{array}
$$

and

$$
\begin{array}{ll}
& f_2 \cdot ((\bot \times id) \times id) \\
\Leftrightarrow & \quad \{\text{definition of } f_2\} \\
& \mathsf{cons} \cdot \langle \overline{\overline{Acc}} \cdot \langle \langle \pi_1 \cdot \pi_2 \cdot \pi_1, \pi_1 \cdot \pi_1 \cdot \pi_1 \rangle, \pi_2 \cdot \pi_2 \cdot \pi_1 \rangle, accounts \cdot \pi_2 \rangle \cdot ((\bot \times id) \times id) \\
\Leftrightarrow & \quad \{\text{fusion-}\times\} \\
& \mathsf{cons} \cdot \langle \overline{\overline{Acc}} \cdot \langle \langle \pi_1 \cdot \pi_2 \cdot \pi_1, \pi_1 \cdot \pi_1 \cdot \pi_1 \rangle, \pi_2 \cdot \pi_2 \cdot \pi_1 \rangle \cdot ((\bot \times id) \times id), accounts \cdot \pi_2 \cdot ((\bot \times id) \times id) \rangle \\
\Leftrightarrow & \quad \{\text{corolary of the cancelation law for } \times, \text{ fusion-}\times\} \\
& \mathsf{cons} \cdot \langle \overline{\overline{Acc}} \cdot \langle \langle \pi_1 \cdot \pi_2 \cdot \pi_1, \pi_1 \cdot \pi_1 \cdot \pi_1 \rangle \cdot ((\bot \times id) \times id), \pi_2 \cdot \pi_2 \cdot \pi_1 \cdot ((\bot \times id) \times id) \rangle, accounts \cdot id \cdot \pi_2 \rangle \\
\Leftrightarrow & \quad \{\text{corolary of the cancelation law for } \times, \text{ fusion-}\times, \text{ identity }\} \\
& \mathsf{cons} \cdot \langle \overline{\overline{Acc}} \cdot \langle \langle \pi_1 \cdot \pi_2 \cdot \pi_1 \cdot ((\bot \times id) \times id), \pi_1 \cdot \pi_1 \cdot \pi_1 \cdot ((\bot \times id) \times id) \rangle, \pi_2 \cdot \pi_2 \cdot (\bot \times id) \cdot \pi_1 \rangle, accounts \cdot \pi_2 \rangle \\
\Leftrightarrow & \quad \{\text{corolary of the cancelation law for } \times, \text{ fusion-}\times, \text{ identity }\} \\
& \mathsf{cons} \cdot \langle \overline{\overline{Acc}} \cdot \langle \langle \pi_1 \cdot \pi_2 \cdot (\bot \times id) \cdot \pi_1, \pi_1 \cdot \pi_1 \cdot (\bot \times id) \cdot \pi_1 \rangle, \pi_2 \cdot \pi_2 \cdot \pi_1 \rangle, accounts \cdot \pi_2 \rangle \\
\Leftrightarrow & \quad \{\text{corolary of the cancelation law for } \times, \text{ identity and } ! \} \\
& \mathsf{cons} \cdot \langle \overline{\overline{Acc}} \cdot \langle \langle \pi_1 \cdot \pi_2 \cdot \pi_1, \pi_1 \cdot \bot \rangle, \pi_2 \cdot \pi_2 \cdot \pi_1 \rangle, accounts \cdot \pi_2 \rangle
\end{array}
$$

Finally, one gets f'

$$
\begin{aligned}
f' &= (\!|[\underline{Sys\ \mathsf{nil}\ \mathsf{nil}}, \overline{Sys} \cdot \langle f'_1, f'_2 \rangle]|\!) \\
f'_1 &= \mathsf{cons} \cdot \langle \overline{Clt} \cdot \bot, clients \cdot \pi_2 \rangle \\
f'_2 &= \mathsf{cons} \cdot \langle \overline{\overline{Acc}} \cdot \langle \langle \pi_1 \cdot \pi_2 \cdot \pi_1, \pi_1 \cdot \bot \rangle, \pi_2 \cdot \pi_2 \cdot \pi_1 \rangle, accounts \cdot \pi_2 \rangle
\end{aligned}
$$

Slice f' is then encoded in HASKELL removing subexpressions marked by $\bot$. This corresponds to removing from the original `migration` code the fragments marked in red. Notice they are exactly the ones processing the client information component in the input sequence.

```
initSystem :: [((CltId, CltName), (AccId, Amount))] -> System
initSystem =
    foldr (curry $ (uncurry Sys) . split f1 f2)
          (Sys [] [])
    where f1 = uncurry (:) . split (uncurry Clt . p1 . p1)
                                   (clients . p2)
          f2 = uncurry (:) . split ((uncurry . uncurry $ Acc) . split (split (p1 . p2 . p1)
                                                                             (p1 . p1 . p1))
                                                                      (p2 . p2 . p1))
                                   (accounts . p2)
```

6 Conclusions and Future Work

This paper presented an approach to slicing of functional programs in which slice identification is regarded as a calculation carried on within the algebra of programming known as the Bird-Meertens calculus. The approach is semantically sound in the sense that the original program may be formally recovered from the collection of its slices, a fact that can only be conjectured in classical, graph-based, syntax-oriented slicing techniques [14, 18]. Moreover it seems to have some potential for practical application as a subsidiary method to such slicing techniques [16]. In a sense, projection and hiding functions encode slicing criteria strongly oriented towards datatypes programs consume or produce. It should be remarked that the idea of using functions to slice other functions can be traced back to [15] where it was used in the context of *regular tree grammars*.

The approach sketched here is, however, in its infancy. Future work includes its generalization to programs whose input and/or output is framed in an *alternative* (sum) context. Whether it scales up to real, heavy examples is currently being assessed by conducting a major case study in foreign open-source HASKELL code.

Another line of research is the application of similar principles to the dual picture of *coinductive* types [10] which enjoy similar calculational properties [17]. This will hopefully lead to a method of *process slicing* to deal with information processes encoded as coalgebraic datatypes [2] with applications to the area of reverse engineering of software architectures (in the sense of *e.g.*, [22]).

References

1. J. Backus. Can programming be liberated from the Von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21:613–641, 1978.
2. L. S. Barbosa and J. N. Oliveira. Coinductive interpreters for process calculi. In *Proc. of FLOPS'02*, pages 183–197, Aizu, Japan, September 2002. Springer Lect. Notes Comp. Sci. (2441).
3. R. Bird. *Functional Programming Using Haskell*. Series in Computer Science. Prentice-Hall International, 1998.
4. R. Bird and O. Moor. *The Algebra of Programming*. Series in Computer Science. Prentice-Hall International, 1997.
5. R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO ASI Series F*, pages 3–42. Springer-Verlag, 1987.
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. J. Gibbons. Conditionals in distributive categories. CMS-TR-97-01, School of Computing and Mathematical Sciences, Oxford Brookes University, 1997.
8. T. Hagino. A typed lambda calculus with categorical type constructors. In D. H. Pitt, A. Poigné, and D. E. Rydeheard, editors, *Category Theory and Computer Science*, pages 140–157. Springer Lect. Notes Comp. Sci. (283), 1987.
9. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conf. on Programming Usage, Design and Implementation*, pages 35–46. ACM Press, 1988.
10. B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–159, 1997.
11. G. R. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–279, 1990.
12. E. Manes and A. Arbib. *Algebraic Approaches to Program Semantics*. Texts and Monographs in Computer Science. Springer Verlag, 1986.
13. E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings of the 1991 ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144. Springer Lect. Notes Comp. Sci. (523), 1991.
14. J. Oliveira. Bagatelle in C arranged for VDM SoLo. *Journal of Universal Computer Science*, 7(8):754–781, 2001. Special Issue on *Formal Aspects of Software Engineering*, Colloquium in Honor of Peter Lucas, Institute for Software Technology, Graz University of Technology, May 18-19, 2001).
15. T. W. Reps and T. Turnidge. Program specialization via program slicing. In *Selected Papers from the Internaltional Seminar on Partial Evaluation*, pages 409–429, London, UK, 1996. Springer-Verlag.
16. F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.
17. V. Vene. *Categorical Programming with Inductive and Coinductive Types*. PhD thesis, Faculty of Mathematics, University of Tartu (*Dissertationsa Mathematicae* 23), 2000.
18. G. Villavicencio and J. Oliveira. Formal reverse calculation supported by code slicing. In *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE 2001, 2-5 October 2001, Stuttgart, Germany*, pages 35–46. IEEE Computer Society, 2001.
19. M. Weiser. *Program Slices: Formal, Psychological and Practical Investigatios of an Automatic Program Abstraction Methods*. PhD thesis, University of Michigan, An Arbor, 1979.
20. M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.
21. M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.
22. J. Zhao. Applying slicing technique to software architectures. In *Proc. of 4th IEEE International Conferencei on Engineering of Complex Computer Systems*, pages 87–98, August 1998.