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 industrial success in their reverse application to the analysis of running pre-existing code.

This paper focuses 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.
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 \( \text{id}_A : A \leftarrow A \) or the so-called final functions \( \text{!}_A : 1 \leftarrow A \) whose codomain is the singleton set denoted by 1 and consequently map every
element of \( A \) into the (unique) element of \( 1 \). Elements \( x \in X \) are represented as points, i.e., functions \( x : X \rightarrow 1 \), and therefore function application \( f \, x \) can be expressed by composition \( f \cdot x \).

Functions can be glued in a number of ways which bear 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 \( f \circ g \) for \( f : B \rightarrow C \) and \( g : B \rightarrow A \). Functions with a common domain can be glued through a split \( \langle f, g \rangle \) as shown in the following diagram:

\[
\begin{array}{c}
A \xrightarrow{\pi_1} A \times B \xrightarrow{\pi_2} B
\end{array}
\]

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 \( \pi_1 : A \rightarrow A \times B \) and \( \pi_2 : B \rightarrow A \times B \), called the projections, which satisfy the following property: for any other set \( Z \) and arrows \( f : A \rightarrow Z \) and \( g : B \rightarrow Z \), there is a unique arrow \( \langle f, g \rangle : A \times B \rightarrow Z \), 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:

\[
k = \langle f, g \rangle \iff \pi_1 \cdot k = f \land \pi_2 \cdot k = g
\]

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., \( \langle \pi_1, \pi_2 \rangle = \text{id} \). 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., \( \langle f, g \rangle \cdot z = \langle f \cdot z, g \cdot z \rangle \). 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:

\[
\text{id} = \langle \pi_1, \pi_2 \rangle \equiv \pi_1 \cdot \text{id} = \pi_1 \land \pi_2 \cdot \text{id} = \pi_2
\]

Equation

\[
\langle \pi_1, \pi_2 \rangle = \text{id}_{A \times B}
\]

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

\[
\pi_1 \cdot \langle f, g \rangle = f, \pi_2 \cdot \langle f, g \rangle = g
\]

\[
\langle g, h \rangle \cdot f = \langle g \cdot f, h \cdot f \rangle
\]

\[
\langle i \times j \rangle \cdot \langle g, h \rangle = \langle i \cdot g, j \cdot h \rangle
\]

The same applies to structural equality:

\[
\langle f, g \rangle = \langle k, h \rangle \iff f = k \land g = h
\]

Finally note that the product construction applies not only to sets but also to functions, yielding, for \( f : B \rightarrow A \) and \( g : B' \rightarrow A' \), function \( f \times g : B \times B' \rightarrow A \times A' \) defined as the split \( \langle f \cdot \pi_1, g \cdot \pi_2 \rangle \). This equates 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 $f : B \to A \times C$, there exists a unique $\bar{f} : B^C \to A$, called the curry of $f$, such that $f = \text{ev} \cdot (\bar{f} \times C)$. Diagrammatically,

\[
\begin{array}{ccc}
A & \xrightarrow{f} & A \times C \\
\bar{f} & \downarrow & \bar{f} \times \text{id}_C \\
B^C & \xrightarrow{\text{ev}} & B
\end{array}
\]

i.e.,

$$k = \bar{f} \equiv f = \text{ev} \cdot (k \times \text{id}) \quad (7)$$

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 $\iota_1 : A + B \to A$ and $\iota_2 : A + B \to B$, called the injections, which satisfy the following universal property: for any other set $Z$ and functions $f : Z \to A$ and $g : Z \to B$, there is a unique arrow $[f, g] : Z \to A + B$, usually called the either (or case) of $f$ and $g$, that makes the following diagram to commute:

\[
\begin{array}{ccc}
A & \xrightarrow{\iota_1} & A + B \\
\xrightarrow{f} & & \xrightarrow{[f, g]} \\
& \downarrow & \downarrow \\
& Z & B
\end{array}
\]

Again this universal property can be written as

$$k = [f, g] \equiv k \cdot \iota_1 = f \land k \cdot \iota_2 = g \quad (8)$$

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

$$[f, g] \cdot \iota_1 = f, [f, g] \cdot \iota_2 = g \quad (9)$$

$$[\iota_1, \iota_2] = \text{id}_{X+Y} \quad (10)$$

$$f \cdot [g, h] = [f \cdot g, f \cdot h] \quad (11)$$

Products and sums interact through the following exchange law

$$\langle [f, g], [f', g'] \rangle = \langle [f, f'], [g, g'] \rangle \quad (12)$$

provable by either product (1) or sum (8) universality. The sum combinator also applies to functions yielding $f + g : A' + B' \to A + B$ 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 $(p \to f, g)$, where $p : 2 \to A$ is a predicate. Intuitively, $(p \to f, g)$ reduces to $f$ if $p$ evaluates to true and to $g$ otherwise. The conditional construct is defined as

$$(p \to f, g) = (f, g) \cdot p?$$

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

$$p? = \begin{array}{ccc}
A & \xrightarrow{[\text{id}, p]} & A \times (1 + 1) \\
\xrightarrow{\text{di}} & A \times 1 + A \times 1 & \xrightarrow{\pi_1 + \pi_1} A + A
\end{array}$$
where \( \text{dl} \) is the distributivity isomorphism. The following laws are useful to calculate with conditionals [7].

\[
\begin{align*}
    h \cdot (p \to f, g) &= (p \to h \cdot f, h \cdot g) \quad (13) \\
    (p \to f, g) \cdot h &= (p \cdot h \to f \cdot h, g \cdot h) \quad (14) \\
    (p \to f, g) &= (p \to (p \to f, g), (p \to f, g)) \quad (15)
\end{align*}
\]

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 \( \text{len} : \mathbb{N} \to A^* \). \( A^* \) is an example of an inductive type: its elements are built by one of the following constructors: \( \text{nil} : A^* \to 1 \), which builds the empty list, and \( \text{cons} : A^* \to A \times A^* \), which appends an element to the head of the list. The two constructors are glued by an either \( \text{in} = [\text{nil}, \text{cons}] \) whose codomain is an instance of polynomial functor \( F X = 1 + A \times X \). The algorithm contents of function \( \text{len} \) is exposed in the following diagram:

\[
\begin{array}{c}
1 + A \times \mathbb{N} \xrightarrow{[0, \text{succ} \cdot \pi_2]} \mathbb{N} \\
\text{id + id} \times \text{len} \\
1 + A \times A^* \xrightarrow{\text{in} = [\text{nil}, \text{cons}]} A^*
\end{array}
\]

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

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

\[
\begin{align*}
    ([\text{in}]) &= \text{id} \quad (16) \\
    ([g]) \cdot \text{in} &= g \cdot F ([g]) \quad (17) \\
    f \cdot ([g]) &= ([h]) \iff f \cdot g = h \cdot F f \quad (18) \\
    ([g]) \cdot T f &= ([g] \cdot F (f, \text{id})) \quad (19)
\end{align*}
\]

where \( T \) is the functor that assigns to a set \( X \) the corresponding inductive type for \( F \) (in our example, \( T \mathbb{X} = X^* \)). 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,

\[
f : T(\Pi_{j \in J} B_j) \to R(\Pi_{i \in I} A_i)
\]
A projection function $\pi: T(B_k) \leftarrow T(\Pi_{j \in J} B_j)$ 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:

$$\pi \cdot f = f'$$

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 $\pi$. 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 $\sigma: R(\Pi_{i \in I} A_j) \leftarrow R(\Pi_{i \in I} A_j)$ 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 $\bot: A_k \leftarrow A_k$ which maps any value of type $A_k$ to the bottom element $\bot_{A_k}$ of the corresponding data domain\(^1\). Formally,

$$\bot = A_k \mapsto 1 \mapsto \bot_{A_k} A_k$$

Therefore, hiding function $\sigma$ 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:

$$f \cdot \sigma = f'$$

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 $\pi$, 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.

```haskell
wc = wcAux (1,0)
```

```haskell
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.
which is translated into the Bird-Merteen's formalism as

$$\text{wcAux } p \ [\] = p$$

$$\text{wcAux } (\text{lc}, \text{cc}) \ (h: t) =$$

$$\text{if } h = ' \backslash n' \text{ then wcAux } (\text{lc}+1, \text{cc}+1) \ t$$

$$\text{else wcAux } (\text{lc}, \text{cc}+1) \ t$$

which is translated into the Bird-Merteen's formalism as

$$\text{wcAux } p \ [\] = p$$

$$\text{wcAux } (\text{lc}, \text{cc}) \ (h: t) =$$

$$\text{if } h = ' \backslash n' \text{ then wcAux } (\text{lc}+1, \text{cc}+1) \ t$$

$$\text{else wcAux } (\text{lc}, \text{cc}+1) \ t$$

which is translated into the Bird-Merteen's formalism as

$$\text{wcAux } p \ [\] = p$$

$$\text{wcAux } (\text{lc}, \text{cc}) \ (h: t) =$$

$$\text{if } h = ' \backslash n' \text{ then wcAux } (\text{lc}+1, \text{cc}+1) \ t$$

$$\text{else wcAux } (\text{lc}, \text{cc}+1) \ t$$

where $$p = (\text{''\backslash n'' } = = ) \cdot \pi_1$$.  

The original function counts the number of lines and characters for a given text input. Our goal is to identify a slice of $$\text{wc}$$ which just computes the number of lines. This is given by the first component of the pair returned by the original $$\text{wc}$$ 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 $$\pi_1 = \text{fst} :: (\text{a}, \text{b}) \rightarrow \text{a}$$, the HASKELL first projection of a pair. Thus the slicing problem reduces to solving the following equation:

$$f' = \pi_1 \cdot \langle \text{[(\text{\text{success}} \times \text{success}) \cdot \pi_2, (\text{id} \times \text{success}) \cdot \pi_2]}} \cdot p?\rangle_F$$

which leads to the following calculation

$$f' = \pi_1 \cdot \langle \text{[(\text{\text{success}} \times \text{success}) \cdot \pi_2, (\text{id} \times \text{success}) \cdot \pi_2]}} \cdot p?\rangle_F$$

$$\text{composing with id}$$

$$\pi_1 \cdot \langle \text{[(\text{\text{success}} \times \text{success}) \cdot \pi_2, (\text{id} \times \text{success}) \cdot \pi_2]}} \cdot p?\rangle_F$$

$$\text{reflexion-× (twice)}$$

$$\pi_1 \cdot \langle \text{[(\text{\text{success}} \times \text{success}) \cdot \langle \pi_1, \pi_2\rangle, (\text{id} \times \text{success}) \cdot \langle \pi_1, \pi_2\rangle]}} \cdot p?\rangle_F$$

$$\text{absorption-× (twice)}$$

$$\pi_1 \cdot \langle \text{[(\text{\text{success}} \times \text{success}) \cdot \pi_2, (\text{id} \times \text{success}) \cdot \pi_2]}} \cdot p?\rangle_F$$

$$\text{fusioin-× (twice)}$$

$$\pi_1 \cdot \langle \text{[(\text{\text{success}} \times \text{success}) \cdot \pi_2, (\text{id} \times \text{success}) \cdot \pi_2]}} \cdot p?\rangle_F$$

$$\text{exchange law, identities}$$

$$\pi_1 \cdot \langle \text{[(\text{\text{success}} \times \text{success}) \cdot \pi_2, (\text{id} \times \text{success}) \cdot \pi_2]}} \cdot p?\rangle_F$$

$$\text{fusioin-× (twice)}$$

$$\pi_1 \cdot \langle \text{[(\text{\text{success}} \times \text{success}) \cdot \pi_2, (\text{id} \times \text{success}) \cdot \pi_2]}} \cdot p?\rangle_F$$

$$\text{fusioin-× (twice)}$$

$$\pi_1 \cdot \langle \text{[(\text{\text{success}} \times \text{success}) \cdot \pi_2, (\text{id} \times \text{success}) \cdot \pi_2]}} \cdot p?\rangle_F$$

$$\text{exchange law}$$

$$\pi_1 \cdot \langle \text{[(\text{\text{success}} \times \text{success}) \cdot \pi_2, (\text{id} \times \text{success}) \cdot \pi_2]}} \cdot p?\rangle_F$$

$$\text{absorption-× (twice), cancelation-× (twice)}$$

$$\pi_1 \cdot \langle \text{[(\text{\text{success}} \times \text{success}) \cdot \text{id} \times \langle \pi_1, \pi_2\rangle, (\text{\text{success}} \times \text{success}) \cdot \langle \pi_1, \pi_2\rangle]}} \cdot p?\rangle_F$$

$$\text{reflexion-× (twice), fusion-+ definition of p}}$$

$$\pi_1 \cdot \langle \text{[(\text{\text{success}} \times \text{success}) \cdot \text{id} \times \langle \pi_1, \pi_2\rangle, (\text{\text{success}} \times \text{success}) \cdot \langle \pi_1, \pi_2\rangle]}} \cdot p?\rangle_F$$

$$\text{absorption-× (twice), cancelation-×, reflexion-×}$$

$$\pi_1 \cdot \langle \text{[(\text{\text{success}} \times \text{success}) \cdot \text{id} \times \langle \pi_1, \pi_2\rangle, (\text{\text{success}} \times \text{success}) \cdot \langle \pi_1, \pi_2\rangle]}} \cdot p?\rangle_F$$

$$\text{McCarthay conditional fusion law}$$
\[ \pi_1 \cdot \langle \langle 1, (\text{succ} \cdot \pi_2, \pi_2) \cdot p^2 \cdot (\text{id} \times \pi_1), [0, (\text{succ} \cdot \pi_2)] \cdot \pi_2 \rangle \rangle_F \]

\[ \pi_1 \cdot \langle \langle 1, (\text{succ} \cdot \pi_2, \pi_2) \cdot p \rangle \cdot \text{id} + (\text{id} \times \pi_1), [0, (\text{succ} \cdot \pi_2)] \cdot \pi_2 \rangle \rangle_F \]

\[ \pi_1 \cdot \langle \langle 1, (\text{succ} \cdot \pi_2, \pi_2) \cdot p^2 \rangle \cdot \text{id} + (\text{id} \times \pi_2), [0, (\text{succ} \cdot \pi_2)] \cdot F \pi_2 \rangle \rangle_F \]

\[ \langle \langle 1, (\text{succ} \cdot \pi_2, \pi_2) \cdot p \rangle \rangle_F, \langle [0, \text{succ} \cdot \pi_2] \rangle_F \rangle \]

\[ \langle \langle 1, (\text{succ} \cdot \pi_2, \pi_2) \cdot p \rangle \rangle_F, \langle [0, \text{succ} \cdot \pi_2] \rangle_F \rangle \]

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

\[ \text{wc} = \text{foldr} \]

\[ (\lambda c \rightarrow \text{if } c == '\n' \text{ then succ else id}) \]

\[ 1 \]

or, going pointwise,

\[ \text{wc} = \text{wcAux} 1 \]

\[ \text{wcAux} :: \text{Int} \rightarrow \text{String} \rightarrow \text{Int} \]
\[ \text{wcAux} \ p \ [] \ = \ p \]
\[ \text{wcAux} \ lc \ (h:t) \ = \]

\[ \text{if } h == '\n' \text{ then wcAux lc+1 t} \]
\[ \text{else wcAux lc t} \]

A similar approach could be taken, using \( \pi_2 \) as a slicing criterion, to isolate the character count computation inside \( \text{wc} \).

### 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:

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

\[ \text{data Client} = \text{Clt} \{ \text{cltid :: CltId}, \]
\[ \text{name :: CltName} \} \]

\[ \text{data Account} = \text{Acc} \{ \text{accid :: AccId}, \]
\[ \text{holder :: CltId}, \]
\[ \text{amount :: Amount} \} \]

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

The target program is now function \( \text{migrate} \) which accepts a sequence of tuples, containing information about bank clients and their respective balances and populates datatype \( \text{System} \). The function procedures recursively on the input sequence and is therefore given as a \( \text{catamorphism} \) for a particular ‘gene’. Consider the following HASKELL definition of \( \text{migrate} \) where the underlying catamorphism is encoded in the standard \( \text{foldr} \) 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))
    (accounts . p2)

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

\[ \text{migrate} = \{ ([Sys \text{nil nil}, \text{Sys} \cdot (f_1, f_2)]) \}_f \]
\[ f_1 = \text{cons} \cdot (\text{Clt} \cdot \pi_1 \cdot \pi_1, \text{clients} \cdot \pi_2) \]
\[ f_2 = \text{cons} \cdot (\text{Acc} \cdot (\pi_1 \cdot \pi_2 \cdot \pi_1 \cdot \pi_1 \cdot \pi_1 \cdot \pi_1 \cdot \pi_2 \cdot \pi_2 \cdot \pi_1), \text{accounts} \cdot \pi_2) \]

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

\[ ([g]) = \text{clients} \cdot ([Sys \text{nil nil}, \text{Sys} \cdot (f_1, f_2)]) \quad (24) \]

Following by calculation over the \( f \) value, one as

\[ ([g]) = \text{clients} \cdot ([Sys \text{nil nil}, \text{Sys} \cdot (f_1, f_2)]) \]
\[ \leftarrow \{ \text{cata-fusion} \} \]
\[ g \cdot (F \text{clients}) = \text{clients} \cdot [Sys \text{nil nil}, \text{Sys} \cdot (f_1, f_2)] \]
\[ \leftarrow \{ \text{constant function and uncurry} \} \]
\[ g \cdot (F \text{clients}) = \text{clients} \cdot [Sys \cdot (\text{nil nil}), \text{Sys} \cdot (f_1, f_2)] \]
\[ \leftarrow \{ \text{absorption-} \} \]
\[ g \cdot (F \text{clients}) = [\text{clients} \cdot (\text{Sys} \cdot (\text{nil nil})), \text{clients} \cdot \text{Sys} \cdot (f_1, f_2)] \]
\[ \leftarrow \{ \text{definition of clients, \cdot is associative} \} \]
\[ g \cdot (F \text{clients}) = [\pi_1 \cdot (\text{nil nil}), \pi_1 \cdot (f_1, f_2)] \]
\[ \leftarrow \{ \text{cancelation-} \times (\text{twice}) \} \]
\[ g \cdot (F \text{clients}) = [\text{nil}, f_1] \]
\[ \leftarrow \{ \text{definition of } f_1 \} \]
\[ g \cdot (F \text{clients}) = [\text{nil cons} \cdot (\text{Clt} \cdot \pi_1 \cdot \pi_1, \text{clients} \cdot \pi_2)] \]
\[ \leftarrow \{ \text{cancelation-} \times (\text{twice}) \} \]
\[ g \cdot (F \text{clients}) = [\text{nil cons} \cdot (\text{Clt} \cdot \pi_1 \cdot \pi_1 \cdot (\pi_1, \text{clients} \cdot \pi_2), \pi_2 \cdot (\pi_1, \text{clients} \cdot \pi_2))] \]
\[ \leftarrow \{ \text{absorption-} \times (\text{twice}) \} \]
\[ g \cdot (F \text{clients}) = [\text{nil cons} \cdot (\text{Clt} \cdot \pi_1 \cdot \pi_1 \cdot (id \times \text{clients}) \cdot (\pi_1, \pi_2), \pi_2 \cdot (id \times \text{clients}) \cdot (\pi_1, \pi_2))] \]
\[ \leftarrow \{ \text{natural-id} \times (\text{twice}), \text{reflection-} \times (\text{twice}) \} \]
\[ g \cdot (F \text{clients}) = [\text{nil cons} \cdot (\text{Clt} \cdot \pi_1 \cdot \pi_1 \cdot (id \times \text{clients}),)] \]
\[ \leftarrow \{ \text{fusion-} \times, \text{def. of functor } F \} \]
\[ g \cdot (id + id \times clients) = [\text{nil}, \text{cons} \cdot (\text{Clt} \cdot \pi_1 \cdot \pi_2)] \cdot (id \times clients) \]

\[
\Rightarrow \quad \{ \text{function equality} \}
\]

\[ g = [\text{nil}, \text{cons} \cdot (\text{Clt} \cdot \pi_1 \cdot \pi_2)] \]

The computed slice is then

\[
[\text{nil}, \text{cons} \cdot (\text{Clt} \cdot \pi_1 \cdot \pi_2)]
\]

which can be translated to Haskell in a straightforward way as

```haskell
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

\[
\sigma = (\bot \times id)^* \tag{25}
\]

Then,

\[
f' = [\text{Sys} nil \text{nil}, \text{Sys} \cdot (f_1, f_2)] \cdot (\bot \times id)^*
\]

\[
\Leftrightarrow \quad \{ \text{cata-absorption} \}
\]

\[
f' = [\text{Sys} nil \text{nil}, \text{Sys} \cdot (f_1, f_2)] \cdot (id + (\bot \times id) \times id)
\]

\[
\Leftrightarrow \quad \{ \text{absorption-+}, \text{idempotency} \}
\]

\[
f' = [\text{Sys} nil \text{nil}, \text{Sys} \cdot (f_1, f_2) \cdot (\bot \times id) \times id]
\]

\[
\Leftrightarrow \quad \{ \text{fusion-} \times \text{(twice)} \}
\]

\[
f' = [\text{Sys} nil \text{nil}, \text{Sys} \cdot (f_1 \cdot (\bot \times id) \times id, f_2 \cdot (\bot \times id) \times id)]
\]

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

\[
f_1 \cdot ((\bot \times id) \times id)
\]

\[
\Leftrightarrow \quad \{ \text{definition of } f_1 \}
\]

\[
\text{cons} \cdot (\text{Clt} \cdot \pi_1 \cdot \pi_2, \text{clients} \cdot \pi_2) \cdot ((\bot \times id) \times id)
\]

\[
\Leftrightarrow \quad \{ \text{fusion-} \times \}
\]

\[
\text{cons} \cdot (\text{Clt} \cdot \pi_1 \cdot ((\bot \times id) \times id), \text{clients} \cdot \pi_2 \cdot ((\bot \times id) \times id))
\]

\[
\Leftrightarrow \quad \{ \text{corollary of the cancelation law for } \times \}
\]

\[
\text{cons} \cdot (\text{Clt} \cdot \pi_1 \cdot ((\bot \times id) \times id), \text{clients} \cdot \pi_2)
\]

\[
\Leftrightarrow \quad \{ \text{corollary of the cancelation law for } \times, \text{idempotency and } ! \}
\]

\[
\text{cons} \cdot (\text{Clt} \cdot \bot, \text{clients} \cdot \pi_2)
\]
and

\[ f_2 \cdot ((\bot \times id) \times id) \]

\[ \Leftrightarrow \{ \text{definition of } f_2 \} \]

\[ \text{cons} \cdot (\overline{\text{Acc}} \cdot (\{\pi_1 \cdot \pi_2 \cdot \pi_1 \cdot \pi_1 \}, \pi_2 \cdot \pi_1), \pi_2 \cdot \pi_1, \pi_2) \cdot ((\bot \times id) \times id) \]

\[ \Leftrightarrow \{ \text{fusion-}x \} \]

\[ \text{cons} \cdot (\overline{\text{Acc}} \cdot (\{\pi_1 \cdot \pi_2 \cdot \pi_1 \cdot \pi_1 \}, \pi_2 \cdot \pi_2 \cdot \pi_1) \cdot ((\bot \times id) \times id), \pi_2 \cdot \pi_2 \cdot \pi_1, \pi_2) \cdot ((\bot \times id) \times id) \]

\[ \Leftrightarrow \{ \text{corollary of the cancelation law for } x, \text{ fusion-}x \} \]

\[ \text{cons} \cdot (\overline{\text{Acc}} \cdot (\{\pi_1 \cdot \pi_2 \cdot \pi_1 \cdot \pi_1 \} \cdot ((\bot \times id) \times id), \pi_2 \cdot \pi_2 \cdot \pi_1, \pi_2) \cdot ((\bot \times id) \times id), \pi_2 \cdot \pi_2 \cdot \pi_1, \pi_2) \]

\[ \Leftrightarrow \{ \text{corollary of the cancelation law for } x, \text{ fusion-}x, \text{ identity} \} \]

\[ \text{cons} \cdot (\overline{\text{Acc}} \cdot (\{\pi_1 \cdot \pi_2 \cdot \pi_1 \cdot (\bot \times id) \times id), \pi_2 \cdot \pi_2 \cdot \pi_1, \pi_2) \cdot (\bot \times id) \times id), \pi_2 \cdot \pi_2 \cdot \pi_1, \pi_2) \]

\[ \Leftrightarrow \{ \text{corollary of the cancelation law for } x, \text{ identity and } ! \} \]

\[ \text{cons} \cdot (\overline{\text{Acc}} \cdot (\{\pi_1 \cdot \pi_2 \cdot \pi_1 \cdot \bot \}, \pi_2 \cdot \pi_2 \cdot \pi_1), \pi_2 \cdot \pi_2 \cdot \pi_1, \pi_2) \]

Finally, one gets \( f' \)

\[
f' = ([\text{Sys} \ \text{nil} \ \text{nil}, \overline{\text{Sys}} \cdot (f'_1, f'_2)]])
\]

\[
f'_1 = \text{cons} \cdot (\overline{\text{Clt}} \cdot \bot, \text{clients} \cdot \pi_2)
\]

\[
f'_2 = \text{cons} \cdot (\overline{\text{Acc}} \cdot (\{\pi_1 \cdot \pi_2 \cdot \pi_1 \cdot \bot \}, \pi_2 \cdot \pi_2 \cdot \pi_1), \pi_2 \cdot \pi_2 \cdot \pi_1, \pi_2)
\]

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.

\[
\text{initSystem} :: [(\text{CltId}, \text{CltName}), (\text{AccId}, \text{Amount})] \rightarrow \text{System}
\]

\[
\text{initSystem} =
\]

\[
\text{foldr} \ (\text{curry} \# \ (\text{uncurry} \ \text{Sys}) \ . \ \text{split} \ f1 \ f2) \ (\text{Sys} \ [[\ ]])
\]

\[
\text{where} \ f1 = \text{uncurry} \ (\cdot) \ . \ \text{split} \ ((\text{uncurry} \ \text{Clt} \ . \ \pi1 \ . \ \pi1) \ (\text{clients} \ . \ p2)
\]

\[
f2 = \text{uncurry} \ (\cdot) \ . \ \text{split} \ ((\text{uncurry} \ . \ \text{uncurry} \# \ \text{Acc}) \ . \ \text{split} \ (\text{split} \ (\pi1 \ . \ p2 \ . \ p1) \ (\pi1 \ . \ p1 \ . \ p1)) \ (\pi2 \ . \ p2 \ . \ p1))
\]

\[
(\pi2 \ . \ p2 \ . \ p1)
\]

(\pi2 \ . \ p2 \ . \ p1)

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