Program Slicing by Calculation*

Nuno F. Rodrigues[1], **Luís S. Barbosa**[1]

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

{nfr,lsb}@di.uminho.pt

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 oriented towards the imperatice or object 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.

This paper focus on a particular program understanding technique — called *code slicing* [20, 18, 19] — which is reframed as a calculational problem in the *algebra of programming* [4]. More specifically, 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 Weiser's thesis [18], 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

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

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 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 code fragments on which the slicing criterion depends, and *forward* slicing [9] which seeks for what depends on or is affected by it.

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. The process is driven by the denotational semantics of the target program, as opposed to more classical syntax-oriented approaches documented in the literature. To make calculation effective and concise we adopt the *pointfree* style of expression [4] popularized among the functional programming community.

This approach seems to be particularly suited to the analysis of *functional* programs. Actually, it offers a way of going inside function definitions and, in some cases, to extract new functions with a restricted input or output. Note that through approaches based on dependencies' graphs one usually works at an 'external' level, for example collecting references to an identifier or determining which functions make use of a particular reference. A recent paper by the authors [15] explore such graphs to identify *components* in functional legacy code. Here, however, we take a completely different path.

The paper is organised as follows. Section 2 discusses the main intuitions behind our approach, characterizing, in particular *backward* and *forward* slicing as calculational problems. The following section contains the main contribution: a case study on slicing by calculation *inductive* functions. A number of concrete examples are discussed. Finally section 4 concludes and points some directions for future work. In a brief appendix, the basic constructions and laws of programming with functions are recalled for reference.

2. Slicing Equations

2.1. Algebra of Programming

In his Turing Award lecture J. Backus [2] 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 [11] 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, 10, 12] and the foundational work of T. Hagino [8].

In this paper we intend to build on this collection of *programming laws* to solve what we shall call *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 [13, 17] its role is compared to one played by the Laplace transform to solve differential equations in a linear space. Appendix A provides a quick introduction to the pointfree algebra of functional programs.

2.2. Slicing Equations

Our starting point is a very simple idea: to identify the 'component' of a function $\Phi : A \longleftarrow B$ affected by a particular argument or contributing to a particular result all one has to do is to *pre*- or *post*-compose Φ with an appropriate function, respectively. In the first case the contribution of an argument is propagated through the body of Φ, forgetting about the role of other possible arguments: σ is a called a *hiding* function and equation

$$\Phi \cdot \sigma \;=\; \Phi' \tag{1}$$

captures the *forward* slicing problem. Φ' is the forward slice of Φ wrt to *slicing criterion* σ. The dual problem corresponds to *backward* slicing: an output, selected through some sort of projection π, is traced back through the body of Φ. The equation is

$$\pi \cdot \Phi \;=\; \Phi' \tag{2}$$

How far can this simple idea be pushed? The simplest case arises whenever Φ is canonical, *i.e.*, defined as an *either* or a *slpit*. In the first case one gets $\Phi = [f, g] : A \longleftarrow B_1 + B_2$. The slicing criterion is simply an embedding, *e.g.*, $\iota_1 : B_1 + B_2 \longleftarrow B_1$ and the forward slice becomes just

$$[f, g] \cdot \iota_1 \;=\; f \tag{3}$$

Dually, for $\langle f, g\rangle : A_1 \times A_2 \longleftarrow B$, one may compute a *backward* slice, by post-composition with a projection, *e.g.*, $p1 : A_1 \longleftarrow A_1 \times A_2$ and conclude

$$\pi_1 \cdot \langle f, g\rangle \;=\; f \tag{4}$$

The dual cases of computing a forward slice of a function with a *multiplictive* domain or a backward slice of a function with a *additive* codomain, amount to composing Φ with the *relational converses* of a projection or an embedding, respectively, leading to equations $\Phi \cdot \pi_1^\circ$ or $\iota_1^\circ \cdot \Phi$. Clearly this is relational composition. From a formal point of view this entails the need to pursue calculation in the *relational* calculus [1]. For the language engineer, however, this means that, in the general case, there is no unique solution to the slicing problem: one may end with a set of possible slices, corresponding to different views over the 'theorectical', relational, non executable, slice.

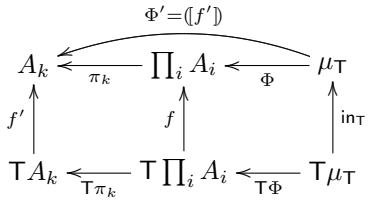
We will not explore this relational counterpart in this paper. Instead our aim is to discuss how far one can go keeping within the functional paradigm in analysing slicing of particularly important class of functions: the *inductive* ones. I.e., functions whose domain is the carrier of an initiall algebra for a regular functor, usually called an *inductive type*. This means that our target functions will be always given by *catamorphisms* [4], *i.e.*, $\Phi = (\!|f|\!)_{\mathsf{T}} : A \longleftarrow \mu_{\mathsf{T}}$, where μ_{T} is the inductive type for functor T and $f : A \longleftarrow \mathsf{T}A$ is the recursion *gene* algebra. Such will be our case-study through the following section.

3. Slicing Inductive Functions

This section is organised around four different slicing cases whose target is always an inductive function $\Phi : A \longleftarrow \mu_{\mathsf{T}}$. Each subsection discusses one of these cases: *product backward*, *sum forward*, *sum backward* and *product forward* slicing. An example is provided in each case.

3.1. Product Backward Slicing

This a 'well-behaved' case: the codomain of Φ is a product and, therefore, the slicing criterion is just the appropriate projection. As Φ is recursive, however, the solution to the slicing problem should be a new *gene* algebra f' such that $\pi_k \cdot \Phi = (\!|f'|\!)$, as explained in the following diagram.



Solving the slicing equation $\Phi' = \pi_k \cdot \Phi$ reduces, by the fusion law for catamorphisms, to verify the commutativity of the leftmost square. This becomes quite clear through an example.

Example. 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)
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 $p = ((' \backslash n' ==) \cdot \pi_1)$ and $\mathsf{F}X = \mathbf{1} + \mathtt{String} \times X$ is the relevant functor.
Our goal is to identify a slice of `wc` which just computes the number of lines. This value is given by the first component of the pair returned by the original `wc` program. Thus, it is expectable that a function which selects the first element of a pair constitutes a good candidate for a slicing criterion. Thus the slicing problem reduces to solving the following equation:

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

which is done as follows

$$
\begin{array}{ll}
 & (\!|f'|\!)_{\mathsf{F}} \;=\; \pi_1 \cdot (\!|[\langle \underline{1}, \underline{0}\rangle, [(succ \times succ) \cdot \pi_2, (id \times succ) \cdot \pi_2] \cdot p?]|\!)_{\mathsf{F}} \\
\Leftarrow & \quad \{\text{cata-fusion}\} \\
 & f' \cdot \mathsf{F}\ \pi_1 \;=\; \pi_1 \cdot [\langle \underline{1}, \underline{0}\rangle, [(succ \times succ) \cdot \pi_2, (id \times succ) \cdot \pi_2] \cdot p?] \\
\Leftrightarrow & \quad \{\text{absorption-}+\text{, cancelation-}\times\text{, natural-}id\text{, definition of }\times\} \\
 & f' \cdot \mathsf{F}\ \pi_1 \;=\; [\underline{1}, [succ \cdot \pi_1 \cdot \pi_2, \pi_1 \cdot \pi_2] \cdot p?] \\
\Leftrightarrow & \quad \{\text{definition of }\times\text{, cancelation-}\times\} \\
 & f' \cdot \mathsf{F}\ \pi_1 \;=\; [\underline{1}, [succ \cdot \pi_2 \cdot (id \times \pi_1), \pi_2 \cdot (id \times \pi_1)] \cdot p?] \\
\Leftrightarrow & \quad \{\text{absorption-}+\text{, } p = p \cdot (\mathsf{id} \times \pi_1)\text{, definiton of }\times\text{, cancelation.}\times\} \\
 & f' \cdot \mathsf{F}\ \pi_1 \;=\; [\underline{1}, [succ \cdot \pi_2, \pi_2] \cdot (id \times \pi_1 + id \times \pi_1) \cdot (p \cdot (id \times \pi_1))?] \\
\Leftrightarrow & \quad \{\text{predicate fusion}\} \\
 & f' \cdot \mathsf{F}\ \pi_1 \;=\; [\underline{1}, [succ \cdot \pi_2, \pi_2] \cdot p? \cdot (id \times \pi_1)] \\
\Leftrightarrow & \quad \{\text{natural-}id\text{, absortion-}+\text{, F definition}\} \\
 & f' \cdot (id + id \times \pi_1) \;=\; [\underline{1}, [succ \cdot \pi_2, \pi_2] \cdot p?] \cdot (id + id \times \pi_1) \\
\Leftrightarrow & \quad \{id + id \times \pi_1 \text{ is surjective}\} \\
 & f' \;=\; [\underline{1}, [succ \cdot \pi_2, \pi_2] \cdot p?]
\end{array}
$$

This calculation leads to the identification of *gene* algebra of the intended slice, which translated back to HASKELL, yields

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

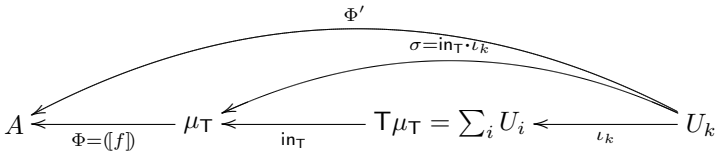
or, going pointwise,

```
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
```

3.2. Sum Forward Slicing

This is also a 'well-behaved' case, in which the slicing criterion reduces to an embedding. The slicing problem, however, requires to be rephrased so that the domain of Φ becomes a sum. This is shown in the following diagram where the slicing criterion is $\sigma = \mathsf{in}_{\mathsf{T}} \cdot \iota_k$, *i.e.*, the relevant embedding composed with the initial algebra (which is an isomorphism).



The computation of Φ' proceeds by the cancellation law for catamorphisms, as illustrated in the following example.

Example. To illustrate a sum forward slicing calculation consider a pretty printer for a subset of the XML language. We start with a data type encoding XML expressions:

```
data XML = SimpElem String [XML]
         | Elem String [(Att, AttValue)] [XML]
         | Text String
type Att = String
type AttValue = String
```

from which functor $\mathsf{F}\,X = S \times X^* + S \times AS \times X^* + S$ is inferred, where `String` and `[(Att, AttValue)]` are abbreviated to S and AS, respectively. Then consider the pretty printer program:

```
pXML (SimpElem e xmls)  = "<" ++ e ++ ">" ++ nl ++
                          (concat . map pXML $ xmls) ++
                          "</" ++ e ++ ">" ++ nl
pXML (Elem e atts xmls) = "<" ++ e ++ concat (map pAtts atts) ++ ">" ++
                          nl ++ (concat . map pXML $ xmls) ++
                          "</" ++ e ++ ">" ++ nl
pXML (Text t)           = t ++ nl
pAtts (att, attvalue)   = " " ++ att ++ "=\"" ++ attvalue ++ "\""
nl                      = "\n"
```

whose pointfree definition reads

$$\begin{aligned}
pXML &= (\!|[[pSElem, pElem], id \star \underline{nl}]|\!)_{\mathsf{F}} \\
pSElem &= ob \star \pi_1 \star cb \star nl \star concat \cdot \pi_2 \star oeb \star \pi_1 \star cb \star nl \\
pElem &= ob \star \pi_1 \cdot \pi_1 \star concat \cdot map\, pAtts \cdot \pi_2 \cdot \pi_1 \star cb \star nl \star \\
&\quad\ concat \cdot \pi_2 \star oeb \star p1 \cdot \pi_1 \star cb \star nl \\
pAtts &= \underline{\text{" "}} \star \pi_1 \star \underline{\text{" = \""}} \star \pi_2 \star \underline{\text{" \""}}
\end{aligned}$$

where $nl = \underline{\text{" \textbackslash n"}}$, $ob = \underline{\text{" <"}}$, $cb = \underline{\text{" >"}}$, $oeb = \underline{\text{" </"}}$, $f \star g = \overline{++} \cdot \langle f, g \rangle$ is a right associative operator and $\overline{++}$ is the uncurried version of the HASKELL operator for list concatenation. The above pointfree definition may seem complex, but it hopefully becomes clear with the following diagram:

$$\begin{array}{ccc}
XML & \xrightarrow{\quad\mathsf{out}_{\mathsf{F}}\quad} & (S \times XML^* + (S \times AS) \times XML^*) + S \\
{\scriptstyle pXML = (\!|f|\!)_{\mathsf{F}}}\downarrow & & \downarrow{\scriptstyle (id \times (\!|f|\!)^*_{\mathsf{F}} + (id \times id) \times (\!|f|\!)^*_{\mathsf{F}}) + id} \\
A & \xleftarrow[f = [[pSElem, pElem], id \star \underline{nl}]]{} & (S \times A^* + (S \times AS) \times A^*) + S
\end{array}$$

Now lets suppose one wants to compute a slice with respect to constructor `SimpElem` of the `XML` data type. This amounts to isolate the parts of the pretty printer that deal with `SimpElem` constructed values. To begin with, one has to define a slicing criterion that isolates arguments of the desired type. This is, of course, given by $\iota_1 \cdot \iota_1$ composed with the initial algebra of the underlying functor, *i.e.*, $\sigma = \mathsf{in}_{\mathsf{F}} \cdot \iota_1 \cdot \iota_1$. The calculation proceeds by cancellation in order to identify the impact of σ over $pXML$.

$$\begin{array}{ll}
 & pXML \cdot \sigma \\
\Leftrightarrow & \quad \{\text{definition of } pXML, \text{ definition of } \sigma\} \\
 & (\!|[[pSElem, pElem], id \star nl]|\!)_{\mathsf{F}} \cdot in_{\mathsf{F}} \cdot (\iota_1 \cdot \iota_1) \\
\Leftrightarrow & \quad \{\text{cata-cancelation}\} \\
 & [[pSElem, pElem], id \star nl] \cdot \mathsf{F}_{pXML} \cdot (\iota_1 \cdot \iota_1) \\
\Leftrightarrow & \quad \{\text{definiton of } \mathsf{F}\}
\end{array}$$

$$
\begin{array}{ll}
 & [[pSElem, pElem], id \star nl] \cdot ((id \times pXML^* + (id \times id) \times pXML^*) + id) \cdot (\iota_1 \cdot \iota_1) \\
\Leftrightarrow & \quad \{\text{definition of } +, \text{ cancelation-}+\} \\
 & [[pSElem, pElem], id \star nl] \cdot (\iota_1 \cdot (\iota_1 \cdot (id \times pXML^*)) \\
\Leftrightarrow & \quad \{\text{cancelation-}+ \text{ (twice)}\} \\
 & pSElem \cdot (id \times pXML^*) \\
\Leftrightarrow & \quad \{\text{definition of } pSElem, \text{ result (5), constant function }\} \\
 & ob \star \pi_1 \cdot (id \times pXML^*) \star cb \star nl \star concat \cdot \pi_2 \cdot (id \times pXML^*) \star \\
 & oeb \star \pi_1 \cdot (id \times pXML^*) \star cb \star nl \\
\Leftrightarrow & \quad \{\text{definition of } \times, \text{ cancelation-}\times\} \\
 & ob \star \pi_1 \star cb \star nl \star concat \cdot pXML^* \cdot \pi_2 \star oeb \star \pi_1 \star cb \star nl
\end{array}
$$

The following result has been used in this calculation

$$(f \star g) \cdot h = f \cdot h \star g \cdot h \tag{5}$$

which is proved as follows:

$$
\begin{array}{ll}
 & (f \star g) \cdot h \\
\Leftrightarrow & \quad \{\text{definition of } \star\} \\
 & \overline{++} \cdot \langle f, g \rangle \cdot h \\
\Leftrightarrow & \quad \{\text{fusion-}\times\} \\
 & \overline{++} \cdot \langle f \cdot h, g \cdot h \rangle \\
\Leftrightarrow & \quad \{\text{definition of } \star\} \\
 & f \cdot h \star g \cdot h
\end{array}
$$

The computed slice is a specialized version of function `pXML`, which only deals with values built with `SimpleElem`. Such function can be directly translated to HASKELL, yielding the following program

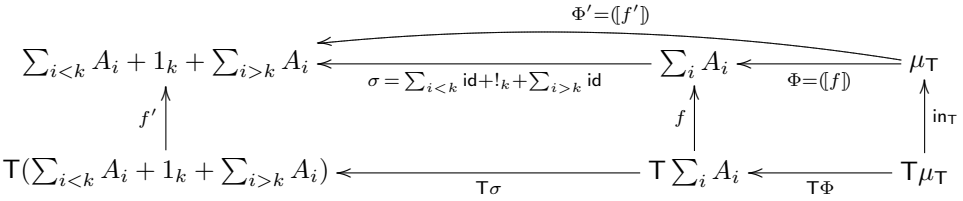
```
pXML' (SimpElem e xmls)  = "<" ++ e ++ ">" ++ nl ++
                           (concat . map pXML \$ xmls) ++
                           "</" ++ e ++ ">" ++ nl
```

3.3. Sum Backward Slicing

The third case is similar to the first one in the sense that in both of them one seeks for backward slices. This time, however, the domain of the original function $\Phi : \sum_i A_i \longleftarrow \mu_{\mathsf{T}}$ is a sum: each slice will therefore be a function which produces values over a specific output type. This complicates the picture: we simply cannot *project* such value from the output of Φ. Just the opposite, the natural slicing criteria would be the converse of a projection.

Let us take a different approach: if projecting is impossible, we may still *hide*. I.e., using the universal $! : 1 \longleftarrow A_k$ to reduce to **1** the output components one wants to get rid of. Hiding functions are constructed by combining $+$, $\times$ and identities with !. Note that in this formulation the slicing criterion becomes *negative* — it specifies what is to be discarded. A we are dealing with inductive functions, the problem is again to find the *gene* for the slice, as documented in the

following diagram.



This sort of slicing is particularly useful when the codomain of original Φ is itself an inductive type, say for a functor G. In such a case one has to compose Φ with the converse of the G-initial algebra in order to obtain an explicit sum in the codomain, *i.e.*

$$\sigma \;=\; (\sum_{i<k} \mathsf{id} + !_k + \sum_{i>k} \mathsf{id}) \cdot \mathsf{out_G}$$

Such is the case discussed in the following example.

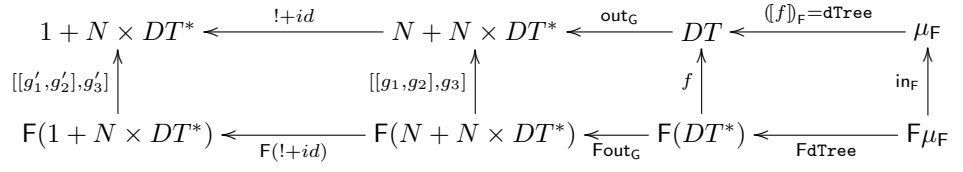
Example. Consider a program which generates the DOM tree of the (simplified) XML language introduced in the previous example. Let F be the corresponding polynomial functor. Note that DOM tree are themselves values of an inductive type for a functor $\mathsf{G}\,X \;=\; N + N \times X^*$, as one may extract from the following HASKELL declaration:

```
data DT a = Leaf NType a
          | Node NType a [DT a]
data NType = NText | NElem | NAtt
```

from which N abbreviates `Ntype` $\times$ `a`. The program to be sliced is `dtree` : $\mu_{\mathsf{G}} \longleftarrow \mu_{\mathsf{F}}$, which is written in pointfree style as follows:

```
dTree = cata g
g = either (either g1 g2) (Leaf NText)
g1 = uncurry (Node NElem)
g2 = uncurry (Node NElem) . split (p1 . p1) (g3 . p2 . p1 <++> p2)
g3 = map (Leaf NAtt . uncurry (++) . (id >< ("="++)))
```

Our aim is to calculate its slice wrt values of type $Node$, *i.e.*, to ifdentify the program components which interfere with the production of values of this type. To do so, the slicing criterion must preserve the right hand side of data type `DT` and slice away everything else (in this case just the left hand side). Thus, we end up with $\sigma = (! + id) \cdot \mathsf{out_G}$. The slicing process is illustrated as follows:



The process proceeds by calculating the new genes g_1', g_2' and g_3' which define the desired slice.

$$\begin{array}{ll} & [[g_1', g_2'], g_3'] \cdot (id \times (! + id) + id \times (! \times id) + id) = (! + id) \cdot [[g_1, g_2], g_3] \\ \Leftrightarrow & \quad \{\text{absortion-+, fusion-+}\} \\ & [[g_1' \cdot (id \times (! + id)), g_2' \cdot (id \times (! \times id))], g_3' \cdot id] = [[(! + id) \cdot g_1, (! + id) \cdot g_2], (! + id) \cdot g_3] \end{array}$$

For the sake of brevity, we shall now consider only the first component of this either equality (the remaining cases follow obviously a similar pattern). Thus, our goal is to find g_1' such that

$$g_1' \cdot (id \times (! + id)) = (! + id) \cdot g_1$$

Note, however, that using the right distributivity isomorphism, g_1 can be further decomposed as follows

$$\begin{array}{ccc} S \times (N + N \times DT^*) & \xrightarrow{distr} & S \times N + S \times (N \times DT^*) \\ \downarrow{\scriptstyle g_1} & \swarrow{\scriptstyle [h_1,h_2]} & \\ N + N \times DT^* & & \end{array}$$

and similarly for $g_1' = [h_3, h_4] \cdot distr$. Then,

$$\begin{array}{ll} & [h_3, h_4] \cdot distr \cdot (id \times (! + id)) = (! + id) \cdot [h1, h2] \cdot distr \\ \Leftrightarrow & \quad \{\text{definition of } distr, \text{ fusion-}+\} \\ & [h_3, h_4] \cdot (id \times ! + id \times id)) \cdot distr = [(! + id) \cdot h1, (! + id) \cdot h2] \cdot distr \\ \Leftrightarrow & \quad \{\text{absorption-}+\} \\ & [h_3 \cdot (id \times !), h_4 \cdot (id \times id)] \cdot distr = [(! + id) \cdot h1, (! + id) \cdot h2] \cdot distr \end{array}$$

Hence

$$h_3 \cdot (id \times !) = (! + id) \cdot h1 \qquad \text{and} \qquad h_4 \cdot (id \times id) = (! + id) \cdot h2$$

Let us concentrate again in the first equality (the other case is similar), that is,

$$\begin{array}{ccc} S \times 1 & \xrightarrow{h_3} & 1 + N \times DT^* \\ {\scriptstyle id \times !}\uparrow & & \uparrow{\scriptstyle !+id} \\ S \times N & \xrightarrow{h_1} & N + N \times DT^* \end{array}$$

In the most general case, functions to a sum type are conditionals. Therefore, we may assume that $h_3 = p \rightarrow \iota_1 \cdot e_1, \iota_2 \cdot e_2$ and $h_1 = q \rightarrow \iota_1 \cdot d_1, \iota_2 \cdot d_2$, respectively. Then,

$$\begin{array}{ll} & (p \rightarrow \iota_1 \cdot e_1, \iota_2 \cdot e_2) \cdot (id \times !) = (! + id) \cdot q \rightarrow \iota_1 \cdot d_1, \iota_2 \cdot d_2 \\ \Leftrightarrow & \quad \{\text{conditionl fusion}\} \\ & p \rightarrow \iota_1 \cdot e_1 \cdot (id \times !), \iota_2 \cdot e_2 \cdot (id \times !) = q \rightarrow (! + id) \cdot \iota_1 \cdot d_1, (! + id) \cdot \iota_2 \cdot d_2 \\ \Leftrightarrow & \quad \{\text{cancelation-}+, \text{ natural } id\} \\ & p \rightarrow \iota_1 \cdot e_1 \cdot (id \times !), \iota_2 \cdot e_2 \cdot (id \times !) = q \rightarrow \iota_1 \cdot !, \iota_2 \cdot d_2 \end{array}$$

which amounts to

$$\begin{array}{rcl} p \cdot (id \times !) & = & q \\ e_1 \cdot (id \times !) & = & ! \\ e_2 \cdot (id \times !) & = & d_2 \end{array}$$

What can be concluded from here? First of all $e_1 =!$. Then $p : \mathbb{B} \longleftarrow S$ is derived from $q : \mathbb{B} \longleftarrow S \times N$ as follows

$$p(s) = \mathsf{false} \equiv \bigvee_n q(s, n) = \mathsf{false}$$

Finally $e_2 : N \times DT^* \longleftarrow S$ comes from $d_2 : N \times DT^* \longleftarrow S \times N$. But what is the relation between them? Actually, abstracting from the second argument of d_2 gives rise to a powerset valued function

$$\begin{array}{rcl} \gamma & : & S \to \mathbb{P}(N \times DT^*) \\ \gamma(s) & = & \{d_2(n,s) \mid n \in N \wedge p(n,s)\} \end{array}$$

Therefore e_2 is just a possible *implementation* of γ. This means that the slice is *not* unique: we are again in the relational world. It should be stressed, however, that the advantage of this calculation process is to lead the program analyist as close as possible of the critical details. Or, puting it in a different way, directs the slice construction until human interaction becomes necessary to make a choice.

3.4. Product Forward Slicing

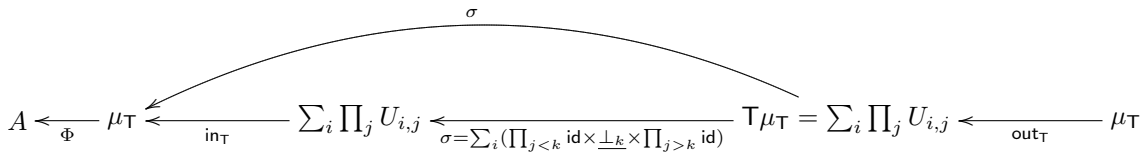
At first sight this is an ackward case as far as inductive functions are concerned. One may resort to out_T to unfold the inductive type, as we did in the sum forward case, but this leads always to a polynomial functor with sums as the main connective. So what do we mean by product forward slicing? Suppose the relevant functor is, say, $\mathsf{F}X = \mathbf{1} + A \times B \times X + B \times X^2$. Our aim is to compute a slice of $\Phi : A \longleftarrow \mu_\mathsf{F}$ corresponding to discarding the contribution of the B component of the parameters.

Our first guess is to adopt the strategy of the previous case and define the slicing criterion as the *hiding* function

$$\mathsf{in}_\mathsf{F} \cdot (\mathsf{id} + \mathsf{id} \times ! \times \mathsf{id} + ! \times \mathsf{id}) : \mu_\mathsf{F} \longleftarrow \mathsf{F}\mu_\mathsf{F}$$

This is *wrong*, of course: the hiding function changes the signature functor. Expression above would become correct if formulated in terms of functor $\mathsf{F}'X = \mathbf{1} + A \times \mathbf{1} \times X + \mathbf{1} \times X^2$. Expression $\mathsf{id} + \mathsf{id} \times ! \times \mathsf{id} + ! \times \mathsf{id}$ becomes a natural transformation from F to F'. However, during calculations the relational converse of this natural transformation would be required and making progress will depend, to a great extent, on the concrete definition of Φ.

Let us, therefore, try a different solution: instead of getting rid of component B, by composition with !, we replace each concrete values by a mark still belonging to B. For that we resort, for the first time in this paper, to the classical semantics of HASKELL in terms of pointed complete partial orders. The qualificative *pointed* means there exist for each type X a bottom element $\perp_X$ which can be used for our purposes as illustrated in the following diagram.



Care should be taken when calculating functional programs in a order-theorectic setting. In particular, as embeddings fail to preserve bottoms, the sum construction is no longer a coproduct and the either is not unique. The set-theorectical harmony, however, can be (almost) recovered if one restricts to *strict* functions (details can be checked in, *e.g.*, [12]). Such is the case of the example below, whose derivation is, therefore, valid.

Example. Let us return to the pretty printer example. Suppose we want to slice away every recursive call in this function. This is achieved by the following slicing criteria $\sigma = in_{\mathsf{F}} \cdot ((id \times \underline{\bot} + (id \times id) \times \underline{\bot}) + id) \cdot out_{\mathsf{F}}$. The calculation proceeds as follows.

$$
\begin{array}{ll}
 & pXML \cdot \sigma \\
\Leftrightarrow & \quad \{\text{definition of } pXML, \text{ definition of } \sigma\} \\
 & (\![[[pSElem, pElem], id \star nl]]\!)_{\mathsf{F}} \cdot in_{\mathsf{F}} \cdot ((id \times \underline{\bot} + (id \times id) \times \underline{\bot}) + id) \cdot out_{\mathsf{F}} \\
\Leftrightarrow & \quad \{\text{cata-cancelation}\} \\
 & [[pSElem, pElem], id \star nl] \cdot \mathsf{F}_{pXML} \cdot ((id \times \underline{\bot} + (id \times id) \times \underline{\bot}) + id) \cdot out_{\mathsf{F}} \\
\Leftrightarrow & \quad \{\text{definiton of } \mathsf{F}, \text{ Functor-}+, \text{ Functor-}\times, \text{ natural-}id\} \\
 & [[pSElem, pElem], id \star nl] \cdot ((id \times (\underline{\bot} \cdot pXML^*) + (id \times id) \times (\underline{\bot} \cdot pXML^*)) + id) \cdot out_{\mathsf{F}} \\
\Leftrightarrow & \quad \{\text{absorption-}+, \text{ natural-}id\} \\
 & [[pSElem \cdot (id \times (\underline{\bot} \cdot pXML^*), pElem \cdot ((id \times id) \times (\underline{\bot} \cdot pXML^*)], id \star nl] \cdot out_{\mathsf{F}}
\end{array}
$$

The calculation continues by evaluating the impact of σ upon each parcel. For the sake of brevity we shall concentrate on the $psElem$ function, other cases being similar. Then,

$$
\begin{array}{ll}
 & pSElem \cdot (id \times (\underline{\bot} \cdot pXML^*) \\
\Leftrightarrow & \quad \{\text{definition of } psElem\} \\
 & ob \star \pi_1 \star cb \star nl \star concat \cdot \pi_2 \star oeb \star \pi_1 \star cb \star nl \cdot (id \times (\underline{\bot} \cdot pXML^*)) \\
\Leftrightarrow & \quad \{\text{constant function, result (5)}\} \\
 & ob \star \pi_1 \star cb \star nl \star concat \cdot \pi_2 \cdot (id \times (\underline{\bot} \cdot pXML^*)) \star oeb \star \pi_1 \star cb \star nl \\
\Leftrightarrow & \quad \{\text{definition of } \times, \text{ cancelation-}\times\} \\
 & cb \star \pi_1 \star cb \star nl \star concat \cdot (\underline{\bot} \cdot pXML^*) \cdot \pi_2 \star oeb \star \pi_1 \star cb \star nl
\end{array}
$$

To recover an executable program, it is necessary to remove from the expression above all occurences of $\underline{\bot}$. Finally, going pointwiase, we obtain the following slice

```
pXML (SimpElem e xmls) = "<" ++ e ++ ">" ++ nl ++ "</" ++ e ++ ">" ++ nl
```

Note, however, that in general, unlike product backward slicing which always yields executable solutions, in this case it may succeed that the final slice is not executable. This does not come to a surprise, since we are filtering input that can be critical to the overall computation of the original function.

4. Conclusions

This paper presented an approach to slicing of functional programs in which slice identification is formulated as an equation in an essentially equational and pointfree program calculus [4].

The requirement that programs should be first translated to a pointfree notation may seem, at first sight, a major limitation. However, automatic translators have been developed within our own research group [6]. Moreover, not only this sort of translators but also rewriting systems to make program calculation a semi-automatia task, are needed to scale up this approach to non academic case-studies. Fortunately this is an active area of research within the algebra of programming community.

Although specific research in slicing of functional programs is sparse, the work of Reps and Turnidge [14] should be mentioned as somewhat related to ours. The ideia of composing

projection functions to slice other functions comes from their work, but the approach they take to analyse the impact of such composition is completely different from ours. They resort to regular tree grammars, which must be previously given in order to compute the desired slices. This way, their approach strictly depends on the actual program syntax. Moreover, they limit themselves to functions dealing with lists or dotted pairs. Another work slightly related to ours is [21] where a functional framework is used to formalize the slicing problem in a language independent way. Nevertheless, their primary goal is not to slice functional programs, but to use the functional *motto* to slice imperative programs given a modular monadic semantics.

The approach outlined in this paper is still in its infancy. Current work includes

- its extension to functions defined by *hylomorphims* [4], with inductive types acting as virtual data structures,
- as well as to the dual picture of *coinductive* functions, *i.e.*, functions to final coalgebras.

This last extension may lead to a method for *process slicing*, with processes encoded in coinductive types (see, *e.g.*, [16] or [3]), with possible applications to the area of reverse engineering of software architectures (in the sense of *e.g.*, [22]).

Finally, we intend to

- to take the *relational* challenge seriously and look for possible gains in calculational power by moving to a category of relations as a preferred semantic universe.

Whether this approach scales up to real, complex examples is currently being assessed by conducting a major case study in foreign open-source HASKELL code.

References

[1] R. C. Backhouse and P. F. Hoogendijk. Elements of a relational theory of datatypes. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development*, pages 7–42. Springer Lect. Notes Comp. Sci. (755), 1993.

[2] 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.

[3] L. S. Barbosa. Process calculi *à la* Bird-Meertens. In *CMCS'01*, volume 44.4, pages 47–66, Genova, April 2001. Elect. Notes in Theor. Comp. Sci., Elsevier.

[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] A. Cunha. *Point-Free Program Calculation*. PhD thesis, Dep. Informática, Universidade do Minho, 2005.

[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] G. R. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–279, 1990.

[11] E. Manes and A. Arbib. *Algebraic Approaches to Program Semantics*. Texts and Monographs in Computer Science. Springer Verlag, 1986.

[12] 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.

[13] 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).

[14] 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.

[15] N. Rodrigues and L. S. Barbosa. Component identification through program slicing. In L. S. Barbosa and Z. Liu, editors, *Proc. of FACS'05 (2nd Int. Workshop on Formal Approaches to Component Software)*, volume (to appear), UNU-IIST, Macau, October 2005. Elect. Notes in Theor. Comp. Sci., Elsevier.

[16] D. Schamschurko. Modeling process calculi with PVS. In *CMCS'98, Elect. Notes in Theor. Comp. Sci.*, volume 11. Elsevier, 1998.

[17] 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.

[18] M. Weiser. *Program Slices: Formal, Psychological and Practical Investigatios of an Automatic Program Abstraction Methods*. PhD thesis, University of Michigan, An Arbor, 1979.

[19] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.

[20] M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

[21] Y. Zhang, B. Xu, and J. E. L. Gayo. A formal method for program slicing. In *2005 Australian Software Engineering Conference (ASWEC'05)*, pages 140–148. IEEE Computer Society, 2005.

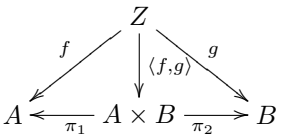
[22] J. Zhao. Applying slicing technique to software architectures. In *Proc. of 4th IEEE International Conferencei on Engineering of Complex Computer Systems*, pages 87–98, August 1998.

A A Glimpse on the Laws of Functions

Composition. This appendix provides a brief review of the algebra of functions, recalling the basic constructions and laws used in the paper. We begin mentioning some functions which 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:

$$k = \langle f, g\rangle \quad \equiv \quad \pi_1 \cdot k = f \ \wedge \ \pi_2 \cdot k = g \tag{6}$$

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 (6) 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

$$\langle \pi_1, \pi_2\rangle \ = \ \mathsf{id}_{A\times B} \tag{7}$$

is called the *reflection* law for products. Similarly the following laws (known respectively as $\times$ *cancelation*, *fusion* and *absorption*) are derivable from (6):

$$\pi_1 \cdot \langle f, g\rangle = f \ , \ \pi_2 \cdot \langle f, g\rangle = g \tag{8}$$

$$\langle g, h\rangle \cdot f \ = \ \langle g \cdot f, h \cdot f\rangle \tag{9}$$

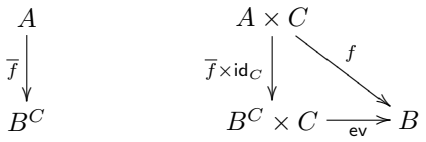
$$(i \times j) \cdot \langle g, h\rangle \ = \ \langle i \cdot g, j \cdot h\rangle \tag{10}$$

The same applies to *structural equality*:

$$\langle f, g\rangle = \langle k, h\rangle \ \equiv \ f = k \ \wedge \ g = h \tag{11}$$

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,

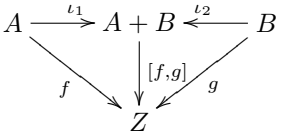


i.e.,

$$k = \overline{f} \quad \equiv \quad f = \mathsf{ev} \cdot (k \times \mathsf{id}) \tag{12}$$

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

$$k = [f, g] \quad \equiv \quad k \cdot \iota_1 = f \ \wedge \ k \cdot \iota_2 = g \tag{13}$$

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

$$[f, g] \cdot \iota_1 = f \ , [f, g] \cdot \iota_2 = g \tag{14}$$

$$[\iota_1, \iota_2] \ = \ \mathsf{id}_{X+Y} \tag{15}$$

$$f \cdot [g, h] \ = \ [f \cdot g, f \cdot h] \tag{16}$$

Products and sums interact through the following *exchange* law

$$[\langle f, g\rangle, \langle f', g'\rangle] \ = \ \langle [f, f'], [g, g']\rangle \tag{17}$$

provable by either product (6) or sum (13) 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 $p : \mathbb{B} \longleftarrow A$ 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

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

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

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

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

$$h \cdot (p \rightarrow f, g) = (p \rightarrow h \cdot f, h \cdot g) \quad (18)$$

$$(p \rightarrow f, g) \cdot h = (p \cdot h \rightarrow f \cdot h, g \cdot h) \quad (19)$$

$$(p \rightarrow f, g) = (p \rightarrow (p \rightarrow f, g), (p \rightarrow f, g)) \quad (20)$$

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:

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

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.*,

$$(\!|in|\!) = \mathsf{id} \quad (21)$$

$$(\!|g|\!) \cdot \mathsf{in} = g \cdot \mathsf{F}\,(\!|g|\!) \quad (22)$$

$$f \cdot (\!|g|\!) = (\!|h|\!) \Leftarrow f \cdot g = h \cdot \mathsf{F}\, f \quad (23)$$

$$(\!|g|\!) \cdot \mathsf{T}\, f = (\!|g \cdot \mathsf{F}\,(f, \mathsf{id})|\!) \quad (24)$$

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.