Paradigmas de Linguagens de Programação: uma abordagem geométrica

Isabel Cafezeiro [*]
Edward Hermann Haeusler [†]
Carlos J. P. Lucena [‡]

PUC-Rio Inf.MCC 21/96 Junh,1996

September 19, 1996

Abstract

This article proposes a framework for defining programming languages paradigms. The main idea is the existence of various levels of abstraction over the notion of programs with inputs. Categorical language and the concept of sheaf are used in order to obtain an approach for a mathematical definition of programming languages paradigms based a geometric definition of computable functions.
Keywords: paradigms, programming languages, categories, sheaf, semantics

Resumo

Este trabalho apresenta uma abordagem para definir paradigmas de linguagens de programação. A idéia central é a existência de vários níveis de abstração sobre a noção de programas com entradas. Utiliza-se linguagem categórica e o conceito de *sheaf* visando a obtenção de uma caracterização matemática de paradigmas de programação baseada na definição geométrica de funções computáveis.
Palavras-chave: paradigmas, linguagens de programação, categorias, feixes, semantica

[*] Depto.Computação UFF(e-mail cafe@dcc.uff.br)
[†] Depto.Informática-PUC-Rio(e-mail hermann@inf.puc-rio.br)
[‡] Depto.Informática-PUC-Rio(e-mail lucena@inf.puc-rio.br)

1 Introdução

Atualmente pode-se constatar que métodos formais vêm sendo usados com bastante freqüência na área de Linguagens de Programação. Este fato é curioso, visto que as linguagens de programação já são, por si próprias, objetos formais: se não o fossem, como poderiam ser executadas pela máquina? Por outro lado, é justamente o fato de serem construídas para 'rodar em uma máquina' que torna necessário o uso de métodos formais em certos estudos da área. A implementação acaba por impor às linguagens de programação certas características que se sobrepoem a conceitos inerentes à linguagem, e, num caso extremo, terminam por camuflar o seu propósito original. Daí, utiliza-se uma metalinguagem - um método formal - que permita resgatar daquela linguagem o aspecto que se quer estudar. Da mesma maneira que as linguagens de programação, os métodos formais têm que ser exatos, mas estes últimos podem conservar, por exemplo, uma sintaxe apropriada ao tipo de investigação pretendida.

Tradicionalmente, a escolha do método formal é feita de acordo com o aspecto que se quer estudar da linguagem. Este fato é importante porque um método particular pode não ser adequado a todos os propósitos. Vamos ilustrar com um exemplo em linguagens imperativas, e em linguagens funcionais.

Semântica Denotacional é um método baseado no conceito matemático de funções. Pode ser usado para dar a semântica de linguagens, entre as quais, linguagens funcionais e imperativas. No entanto, para certos aspectos das linguagens imperativas será necessário fazer alguns ajustes no método. Para dar a semântica do comando GOTO, é preciso acrescentar a noção de *continuation*; para dar a semântica de comandos paralelos, é preciso acrescentar *ressumption*.

Semântica Operacional costuma se baseadar na noção de estado, e transições de estados. Funciona de maneira bastante adequada no tratamento de linguagens imperativas, onde memória e atribuição exercem papéis de maior relevância. No entanto, é preciso investir algum esforço quando se pretende descrever, por transições de estados, a semântica de linguagens funcionais, na qual não são usados os conceitos de memória e atribuição.

Como se vê, se, por um lado, podemos utilizar métodos formais para abstrair das linguagens de programação as características introduzidas pelo modelo de implementação, por outro lado, os próprios métodos formais introduzem, nas descrições das linguagens, aspectos inerentes aos conceitos que os fundamentam.

Com base nestas questões, nosso estudo neste artigo segue na direção de apresentar um mecanismo que seja capaz de descrever diversos paradigmas de linguagens de programação, sem recair no 'paradigma' do método formal. Abstraindo características particulares, tal mecanismo seria mais adequado para compreender, classificar, e até mesmo propor modelos de implementação para linguagens.

Vamos, então, procurar o que há de semelhante nas entidades que fazem parte do 'mundo da programação' - programas, programas instanciados, linguagens, paradigmas, funções computáveis, etc - e caracterizar esta semelhança através de uma linguagem geométrica. Por exemplo, imagine que cada uma destas entidades pode ser representada por um objeto geométrico. Nosso trabalho, aqui, será o de comparar estes objetos, procurando em suas formas algo que permita agrupá-los. O próprio conceito de objeto será definido sobre uma noção geométrica (o conceito de *sheaves*), já que esta noção é bem definida em Teoria das Categorias. Descreveremos objetos em linguagem categórica.

Como pano de fundo para este trabalho, temos a referência [1], na qual a noção de objeto é definida através de *sheaves*. A noção apresentada aqui é inspirada no referido trabalho.

2 A abordagem geométrica

A opção por uma abordagem geométrica, como foi dito anteriormente, baseia-se na necessidade de adoção de uma linguagem que não incorpore características de um ou de outro paradigma particular. Mas esta opção traz consigo a dificuldade de ter que extrair uma forma de algo que parece ser completamente abstrato: o paradigma.

Já que o paradigma, em si, é algo demasiadamente impreciso, vamos, então, partir do que conhecemos de concreto nos paradigmas: as execuções, ou *traces* de programas. É fácil descrever geometricamente o *trace* de um programa: basta ver a curva descrita pela variação de seus componentes, ou atributos. Sendo assim, podemos considerar um domínio espaço-temporal (espaço para retratar os atributos, tempo para retratar cada passo da execução) e um domínio de valores. A dinâmica poderá ser descrita por um mapeamento que designará, a cada localidade no espaço e evolução no tempo, os valores dos atributos para aquela localidade.

A partir daí nossa estratégia será a de aumentar, aos poucos, o nível de abstração. Dados, então, todos os possíveis *traces* de todas as possíveis implementações, vamos agrupá-los por semelhanças geométricas de forma a ter conjuntos de *traces* que representam a masma implementação. O passo seguinte é, então, procurar semelhanças nas descrições geométricas dessas implementações de forma a identificar os casos que computam a mesma função, e finalmente, caracterizar funções computáveis.

O presente trabalho trata da caracterização geométrica de funções computáveis, e descreve as bases de nossa perspectiva futura: a caracterização de paradigma. Esta, tem como diretivas as idéias a seguir.

Tendo baseado a descrição geométrica de funções computáveis em mapeamentos do domínio espaço-temporal em valores, vamos encontrar os paradigmas nas formas de lidar com este domínio. Por exemplo, já que o paradigma funcional tem como característica o fluxo de valores durante a computação, encontraremos as implementações funcionais de uma função computável restringindo o domínio espaço-tempo a uma localidade de tamanho unitário nos espaços. Esta projeção no espaço será capaz de descrever a função através das variações no tempo. Já nas implementações imperativas da mesma função computável, será preciso avaliar uma localidade de tamanho maior, visto que a variação de cada posição de memória exerce um papel fundamental neste paradigma. Nestes casos, a projeção nos espaços é insuficiente para dar a compreensão de toda a função.

A ideia central de nossa abordagem consiste em encarar programas com entrada como se fossem objetos do 'mundo real'. Passamos, então a discutir a noção de objeto.

3 O que caracteriza um objeto?

Nesta seção apresentamos uma série de noções puramente intuitivas a respeito do processo de concepção de um objeto por um indivíduo. Não é nosso propósito fazer, aqui, um estudo técnico do ponto de vista filosófico a este respeito. Ao contrário, pretendemos trabalhar com as idéias mais simples, que nos vêm a mente ao pensarmos neste assunto.

Quando um objeto qualquer é exposto a um indivíduo, este, imediatamente, constrói um conjunto de percepções. Por exemplo, percepções a respeito da cor, do calor, da forma, do cheiro. O conjunto destas percepções associado aos relacionamentos destas com outras percepções mais antigas, é que vão construindo o conceito do objeto para aquele indivíduo. Na verdade, também devemos incluir neste conjunto aquelas percepções que são transmitidas por terceiros, que, por vezes, influem na formação da noção do objeto para o indivíduo. Quando o indivíduo possui algum conhecimento a respeito de um objeto, e, em seguida, recebe novas percepções, estas podem ser simplesmente adicionadas às antigas, ou, no caso de alguma incoerência, podem provocar a revisão de percepções já aceitas.

Há que se considerar ainda as influências provenientes do contexto em que o indivíduo se insere. O que aconteceria, por exemplo, se exibíssemos o mesmo objeto ao mesmo indivíduo em momento e local diferentes? Provavelmente o conjunto de percepções não seria o mesmo, mas algo neste conjunto haveria de se manter, já que se trata do mesmo objeto. Considere agora o mesmo objeto, no mesmo contexto, sendo exposto a dois indivíduos diferentes. Ainda assim, algo haveria de se manter nas percepções individuais. Parece, então que, quando se trata do mesmo objeto, deve existir uma certa coerência entre os conjuntos de observações. Se não há coerência, não se entende o que é o objeto.

Para definir o que é um objeto, vamos tomar conjuntos de observações, ou percepções que concordam em certos aspectos:

Um objeto é um conjunto de observações coerentes.

Cabe agora responder: O que é exatamente uma observação? O que significa ser coerente ?

4 Observações são funções

Uma observação é uma função $f : U \rightarrow A$, onde A é um conjunto de atributos, e U é um aberto de $\mathcal{I}_{0,0}$. Este último é apenas um domínio espaço-temporal. Sem maiores considerações, vamos dizer que $\mathcal{I}_{0,0}$ é um espaço topológico. Mais tarde, voltaremos a esta questão. Considerando tempo e espaço como entidades discretas, teremos:

$$\mathcal{I}_{0,0}(\omega, \omega) = \{T \times S \ : \ T, S \in \mathcal{I}_0(\omega)\},$$

$$\text{com } \mathcal{I}_0(\omega) = \{\emptyset, \{0\}, \{0,1\}, \{0,1,2\}, ...\} \cup \{\omega\}.$$

Vamos desconsiderar o fato de que novas percepções poderiam causar o descarte de percepções mais antigas. Imagine apenas que uma nova percepção pode ser ou não adicionada ou não ao conjunto. Sendo assim, o conjunto de observações que caracteriza o objeto aumenta, ou permanece o mesmo a medida em que o tempo passa. A mesma idéia pode ser aplicada ao espaço. Ao expandirmos a área de observação, o conhecimento a respeito do objeto pode crescer, ou se manter.

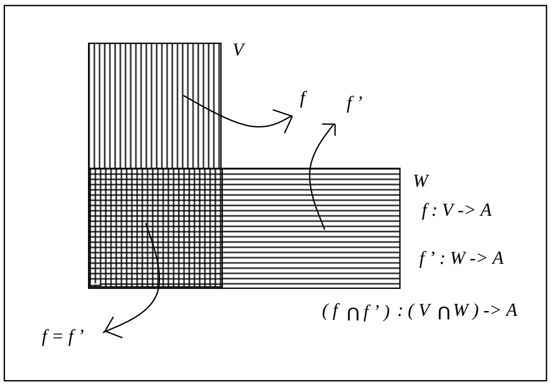
Pela monotonicidade do conjunto de observações, temos a segunte condição de restrição:

$$f = f' \ U,$$
Para $U \preceq V$, abertos em $\mathcal{I}_{0,0}$, e $f' : V \rightarrow A$,
f tem os mesmos valores que f', mas só é definido em U.

Seja $\mathcal{O}(U)$ o conjunto de observações do objeto $\mathcal{O}$ a partir de um domínio U. Então, $\mathcal{O}(U)$ é um conjunto de funções $f : U \rightarrow A$. Passamos, então a definir o que são observações coerentes:

Dados V, W abertos em $\mathcal{I}_{0,0}$ com $V \cap W \neq \emptyset$,
se para todo $f_V \in \mathcal{O}(V)$ e para todo $f_W \in \mathcal{O}(W)$ temos $f_V = f_W$ em $\mathcal{O}(V \cap W)$,
então, podemos dizer que $\mathcal{O}(V)$ e $\mathcal{O}(W)$ são conjuntos de observações coerentes.

Informalmente, queremos que, do ponto de vista do mesmo domínio, tenhamos a mesma visão do objeto, como ilustra a figura a seguir.



Observações coerentes significam que, na interseção dos domínios, temos o mesmo conhecimento a respeito do objeto. Com relação à união, gostaríamos de descrever o conhecimento do objeto na união dos domínios a partir do conhecimento em cada domínio componente, ou seja, a partir da união das funções em cada domínio componente.

É necessário, então, definir a seguinte condição:

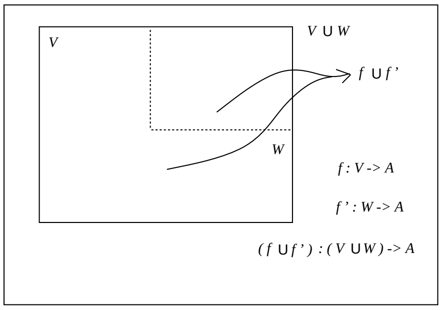
Dados V, W abertos em $\mathcal{I}_{0,0}$ com $V \cap W \neq \emptyset$,
se $\mathcal{O}(V)$ e $\mathcal{O}(W)$ são observações coerentes, com $f \in \mathcal{O}(V)$ e $f' \in \mathcal{O}(W)$,
então, $f \cup f' \in \mathcal{O}(V \cup W)$.

Na definição acima, a união das funções é o menor limite superior da união dos dois grafos, considerando a ordem $f \leq f'$ sse $f' dom(f) = f$.

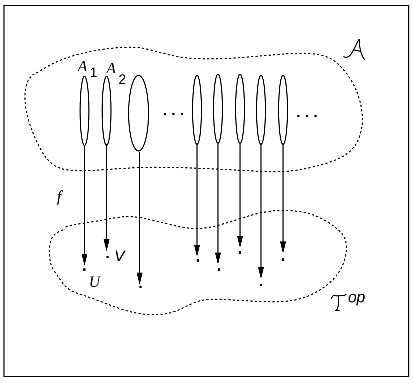
Mas existe um problema com relação a esta definição: estamos considerando o espaço topológico formado por retângulos com canto inferior esquerdo na origem. Mas a união de dois retângulos deste tipo pode não ser um retângulo, e portanto, não pertence ao espaço topológico. Para obter o fechamento sob união arbitrária, vamos redefinir a união como:

$$\bigcup_{\mathcal{I}_n} \quad = \quad < max(I_n), max(J_n) >, \text{ com } \mathcal{I}_n =< I_n, J_n > \text{ e } n \in \omega$$

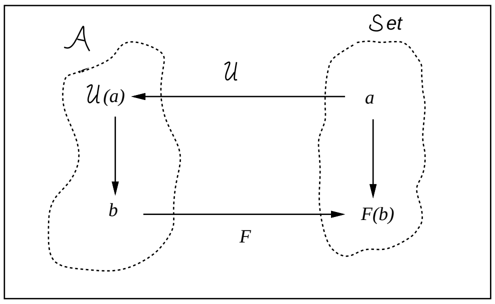
Agora sim, temos que $\mathcal{I}_{0,0}$ é realmente um espaço topológico que chamaremos de $\mathcal{T}$. A figura a seguir ilustra a condição de união.



Em decorrência desta nova definição de união, temos que $< \mathcal{A}, f^{-1} >$ é um feixe (*sheaf*) sobre $\mathcal{T}$, como mostra a figura a seguir.

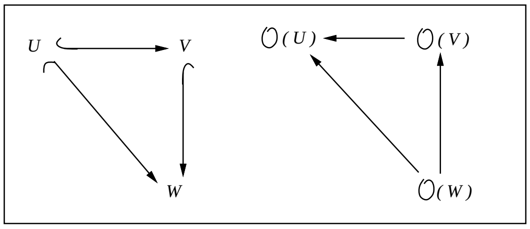


$\mathcal{A}$ é qualquer categoria que tenha um funtor de esquecimento F, responsável por anular qualquer estrutura existente, retratando a categoria em $\mathcal{S}et$, e $\mathcal{U}$, seu funtor adjunto. A existência de ambos permite trabalharmos com $\mathcal{S}et$ no lugar de $\mathcal{A}$.



5 Objetos são funtores contravariantes

Nas seções anteriores definimos $\mathcal{O}(U)$ como o conjunto de funções f que representam observações do objeto $\mathcal{O}$ a partir do domínio U. Pela condição de restrição, temos que, dados U e V, abertos em $\mathcal{T}$, de tal forma que exista uma função de inclusão $\imath : U \hookrightarrow V$, teremos, em $\mathcal{S}et$ um morfismo $\mathcal{O}(\imath) : \mathcal{O}(V) \to \mathcal{O}(U)$. Se considerarmos um outro aberto W em $\mathcal{T}$, com $U \hookrightarrow V \hookrightarrow W$, teremos $\mathcal{O}(W) \to \mathcal{O}(V) \to \mathcal{O}(U)$. Temos, ainda, que $\mathcal{O}(I_U) = \mathcal{O}(U \to U) = \mathcal{O}(U) \to \mathcal{O}(U) = I_{\mathcal{O}(U)}$. Daí, $\mathcal{O}$ é um funtor contravariante: $\mathcal{O} : \mathcal{T}^{op} \to \mathcal{S}et$.



6 Focalizando uma implementação instanciada

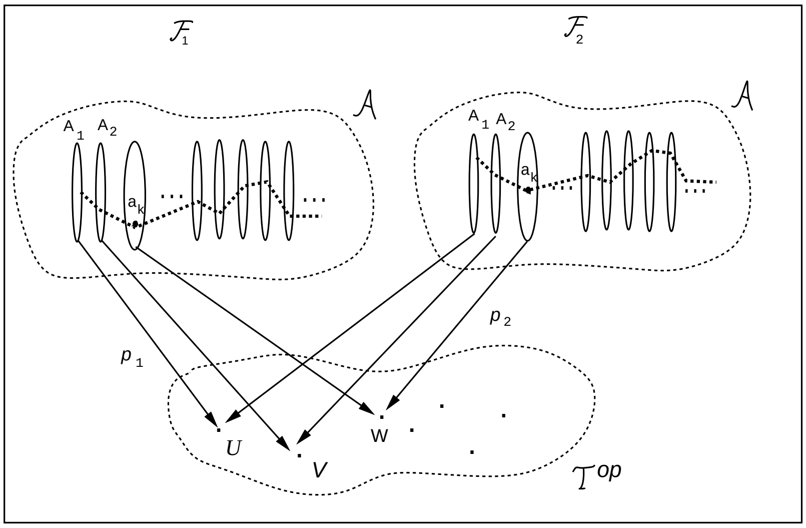
Tendo definido o que é um objeto, podemos, agora, abordar uma implementação instanciada sob a ótica dos objetos. Chamamos de implementação instanciada a uma implementação cujas entradas são definidas. Por exemplo, a implementação Soma(3,2) é uma instância da implementação Soma(x,y), onde se substitui x por 3 e y por 2.

Considere o funtor contravariante $\mathcal{O}$, dos abertos da topologia $\mathcal{T}$, em $\mathcal{S}et$. De um lado, temos $\mathcal{S}et$: conjuntos de atributos. Na verdade, não sabemos ainda o que são exatamente esses atributos, mas vamos considerar que eles representam todos as informações necessárias para descrever o funcionamento de uma implementação instanciada. Do outro lado, temos $\mathcal{T}^{op}$, o espaço topológico que representa espaço-tempo. O funtor $\mathcal{O}$ associa aos abertos de $\mathcal{T}^{op}$ toda a informação que se consegue enxergar daquela localidade. À medida em que aumentamos os abertos com relação ao tempo, podemos ver a variação de cada atributo no tempo. Essa variação obedece à função de restrição: se voltarmos a um tempo passado, retornaremos ao conhecimento daquele momento. O mesmo acontece com relação à variações no espaço. À medida em que aumentamos o espaço das observações, podemos enxergar mais detalhes da implementação: o conjunto de atributos visíveis aumenta. Mas, se retornarmos a um espaço menor, restringimos a visibilidade ao que tínhamos antes. Para cada implementação, existe um tamanho ideal para espaços, a partir do qual pode-se ver todos os atributos relevantes para descrever uma implementação instanciada. Se for considerado um espaço maior do que este, os atributos introduzidos permanecerão com um valor indefinido em todos os tempos. Também com relação aos tempos, se considerarmos programas que páram, existe um aberto ideal, do qual pode-se ver a variação dos atributos durante toda execução da implementação instanciada. Qualquer aberto maior do que este, será mapeado no conjunto vazio, o que caracteriza o fim da execução.

Seja $\mathcal{F}_1$ uma implementação específica. $\mathcal{F}_1$ é um *sheaf* sobre $\mathcal{T}$. Temos, então, uma função $p : A \to \mathcal{T}$, onde A é a união de todos os conjuntos de atributos A_i: $A = \{a : a \in A_i$ para qualquer $i\}$. A função p associa cada A_i a um único aberto em $\mathcal{T}^{op}$.

$$\mathcal{F}_1 = \{p_1^{-1}(U), U \in \mathcal{T}\} = A$$

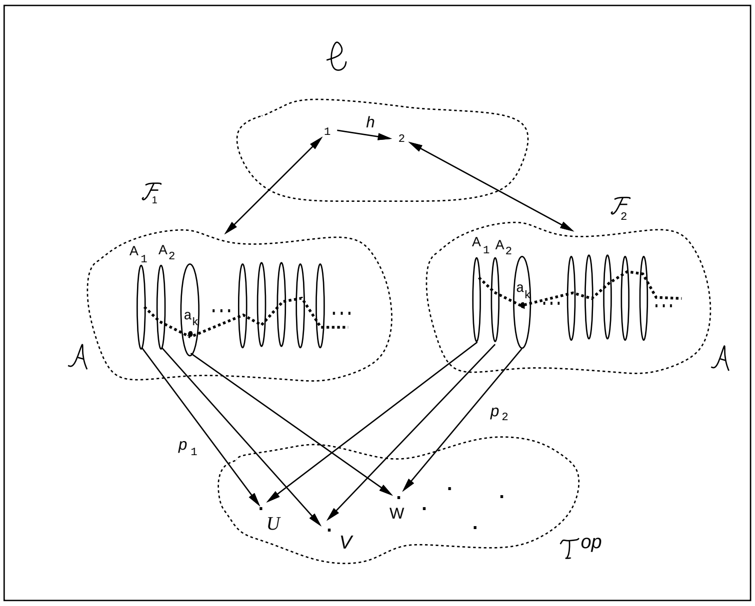
A figura a seguir mostra duas implementações instanciadas $\mathcal{F}_1$ e $\mathcal{F}_2$.



Seções neste *sheaf* são funções localmente homeomórficas $S : \mathcal{T} \rightarrow A$ que associam cada aberto em $\mathcal{T}$ a um elemento a_k, valor de algum atributo em A_i. Podemos visualizar uma seção como a linha tracejada que a figura acima mostra: ela liga um valor a_k de cada A_i. Existem vários tipos de seções em um *sheaf*, mas nos interessam apenas aquelas que ligam um valor em A_i ao valor correspondente ao mesmo atributo em A_j, onde $\mathcal{F}_1(U) = A_i$, $\mathcal{F}_1(V) = A_j$ e $U \hookrightarrow V$.

Seguindo a linha pontilhada de uma destas seções, pode-se acompanhar o comportamento do atributo correspondente como se estivéssemos seguindo passo a passo a execução. Seja a_k o valor de um atributo k de $\mathcal{F}_1$ cujo comportamento é descrito por S_1^k: o índice superior representa o atributo, e o inferior o funtor. Considere um outro funtor $\mathcal{F}_2$, representando a mesma implementação que $\mathcal{F}_1$, mas instanciado com valores diferentes. Haverá também em $\mathcal{F}_2$ uma seção S_2^k representando o comportamento de k. Podemos comparar, passo a passo, através da linha tracejada, os valores que k assume em $\mathcal{F}_1$ e os valores que k assume em $\mathcal{F}_2$. As duas linha tracejadas mantém uma certa semelhança, como se uma fosse apenas uma deformação da outra. Este fato é esperado visto que $\mathcal{F}_1$ e $\mathcal{F}_2$ são a mesma implementação, e o código não se altera durante a execução. Existe, então, para cada atributo k em $\mathcal{F}_1(U)$, uma função que retorna o valor o atributo k correspondente em $\mathcal{F}_2(U)$. Chamaremos de η o conjunto das funções η_{k} para todo k.

Cabe agora investigar qual é o fator que determina a deformação na seção. Este fator é incorporado aos funtores $\mathcal{F}$ por alguma entidade que não seja a implementação, pois esta é invariante. Em $\mathcal{T}^{op}$ também não se encontra este fator, já que a topologia é a mesma para os dois funtores. O motivo da deformação reside, então, numa categoria que chamaremos de $\mathcal{C}$, cujos objetos são as entradas e os morfismos são funções que transformam uma entrada em outra. $\mathcal{C}$ é a categoria das entradas. Seja h o morfismo de $\mathcal{C}$, que transforma a entrada de $\mathcal{F}_1$ na entrada de $\mathcal{F}_2$. Usando h, pode-se transformar S_1 em S_2 para cada U.



Daí, η tem a forma

$$\eta(V) : \mathcal{F}_1(V) \to \mathcal{F}_2(V)$$
$$\eta(V)(S_1(V)) = g(h)(1) = S_2(V)$$

onde 1,2 são as entradas, h é a função que associa as entradas: $h(1) = 2$,
e g só depende da entrada, e de h.

Temos, então que

Para cada atributo k, η é uma transformação natural em $\mathcal{T}^{op}$:
O diagrama seguinte comuta para qualquer U e V.

$$\begin{tikzcd}
\mathcal{F}_{1(U)} \arrow[r, "\eta(U)"] & \mathcal{F}_{2(U)} \\
\mathcal{F}_{1(V)} \arrow[u] \arrow[r, "\eta(V)"'] & \mathcal{F}_{2(V)} \arrow[u]
\end{tikzcd}$$

Prova: As setas verticais são dadas pela função de restrição. As setas horizontais são construídas atributo a atributo, dado que, para cada atributo, ambas as seções variam sobre a mesma topologia, ambas as entradas são conhecidas, e o código é único.

Exemplo: Considere a implementação

```
ler(x,y)
z <- x
repetir y
  z <- z+1
fim
```

e os funtores $\mathcal{F}_1$ e $\mathcal{F}_2$ representando as entradas (3,2) e (1,6). Abaixo, mostramos as *traces* dos dois funtores para os atributos que representam as variáveis x,y e z.

(*tempos*)		0	1	2	3	4	5	6	7	8	...
$\mathcal{F}_1$											
x	:	3	3	3	3	3	3	3	3	3	...
y	:	2	2	2	2	2	2	2	2	2	...
z	:	$\perp$	3	4	5	5	5	5	5	5	...
$\mathcal{F}_2$											
x	:	1	1	1	1	1	1	1	1	1	...
y	:	6	6	6	6	6	6	6	6	6	...
z	:	$\perp$	1	2	3	4	5	6	7	7	...

Vamos selecionar o atributo que representa o comportamento da variável z. Teremos η^{z}:

$$\begin{array}{lcll}
\eta^{\mathrm{z}}(T \times S)(\perp) & = & \perp & \text{se } |T| = 0 \\
\eta^{\mathrm{z}}(T \times S)(3) & = & \pi_1(h(3,2)) & \text{se } |T| = 1 \\
\eta^{\mathrm{z}}(T \times S)(4) & = & g^{|T|-1}(\pi_1(h(3,2))) & \text{se } |T| \leq \pi_2(h(3,2)) \\
\eta^{\mathrm{z}}(T \times S)(5) & = & F(\pi_1(h(3,2)), \pi_2(h(3,2))) & \text{senão}
\end{array}$$

Acima, temos que $h(x,y) = < x-2, y+4 >$ retorna a entrada de $\mathcal{F}_2$ com x, y sendo a entrada de $\mathcal{F}_1$. A letra π é a função de projeção. A função g é determinada pela evolução do atributo z em $\mathcal{F}_1$, que é conhecida por S_1^z. Neste caso, $g(w) = w+1$, conforme mostra a linha 4 do programa. A notação g^n significa n aplicações da função g, e, finalmente, F representa a função computável efetuada por esta implementação. É importante destacar dois detalhes: primeiro, estamos considerando um espaço que faz o atributo z visível. Segundo, haverá no mínimo um atributo em A, para o espaço ideal, onde encontraremos F, a função computável efetuada.

7 Implementação: uma definição categórica

Já definimos o que são implementações instanciadas. Sabemos também de que forma encontrar η, a transformação natural que associa uma implementação instanciada, à mesma implementação, instanciada para outros valores. Definiremos, então:

Uma implementação é a categoria cujos objetos são os funtores $\mathcal{F}$
e os morfismos são conjuntos de transformações naturais η.

Prova: Pela definição de categoria, tem-se:
(i) Objetos da categoria: $\mathcal{F}$, funtores de $\mathcal{T}^{op}$ em A.
(ii) Morfismos da categoria: η, transformações naturais entre os funtores.
(iii) Para cada par $< \eta, \eta' >$, com *cod* η = *dom* η', tem-se $\eta' \circ \eta$, tal que $dom(\eta' \circ \eta) = dom$ η e $cod(\eta' \circ \eta) = cod$ η', e ainda $\eta'' \circ (\eta' \circ \eta) = (\eta'' \circ \eta') \circ \eta$ já que cada quadrado do diagrama abaixo comuta.

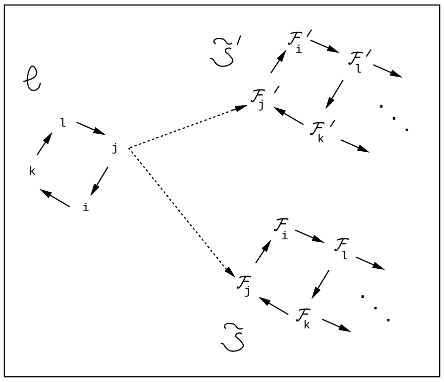
$$\begin{tikzcd}
U \arrow[d, hook, "i"] & \mathcal{F}_1(U) \arrow[r, "\eta"] & \mathcal{F}_2(U) \arrow[r, "\eta'"] & \mathcal{F}_3(U) \arrow[r, "\eta''"] & \mathcal{F}_4(U) \\
V & \mathcal{F}_1(V) \arrow[u, "\mathcal{F}_1(i)"] \arrow[r, "\eta"'] & \mathcal{F}_2(V) \arrow[u, "\mathcal{F}_2(i)"] \arrow[r, "\eta'"'] & \mathcal{F}_3(V) \arrow[u, "\mathcal{F}_3(i)"] \arrow[r, "\eta''"'] & \mathcal{F}_4(V) \arrow[u, "\mathcal{F}_4(i)"]
\end{tikzcd}$$

(iv) Para cada objeto $\mathcal{F}$, define-se $1_{\mathcal{F}}: \mathcal{F} \to \mathcal{F}$ tal que, dados $\eta: \mathcal{F}_1 \to \mathcal{F}$, e $\eta': \mathcal{F} \to \mathcal{F}_2$, temos $1_{\mathcal{F}} \circ \eta = \eta$ e $\eta' \circ 1_{\mathcal{F}} = \eta'$. $1_{\mathcal{F}}$ é a transformação identidade.

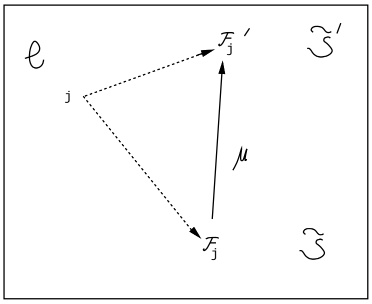
8 A função computável

Tendo caracterizado o que é uma implementação, gostaríamos, agora, de elevar um pouco mais o nível de abstração, e associar implementações diferentes que computam a mesma função. Podemos, então, imaginar a categoria cujos objetos são todas as categorias que representam implementações, e cujos morfismos são quaisquer relacionamentos entre estas implementações. Nossa questão, agora, é como separar deste emaranhado apenas aquelas categorias que computam a mesma função.

É fato que duas implementações que computam a mesma função, mesmo sendo diferentes, têm que retornar o mesmo resultado para a mesma entrada. Sejam $\Im$ e $\Im'$ as categorias que representam essas duas implementações, e $\mathcal{C}$ a categoria das entradas. Existem dois funtores, um de $\mathcal{C}$ em $\Im$, e outro de $\mathcal{C}$ em $\Im'$, que recuperam, em cada categoria o funtor associado àquela entrada. Estes funtores estão ilustrados pelas linhas tracejadas da figura a seguir, onde $\mathcal{F}_i$,$\mathcal{F}_j$,...,$\mathcal{F}'_i$,$\mathcal{F}'_j$,... representam a implementação instanciada para a entrada i, j, ...



Diremos que duas implementações instanciadas computam a mesma função (também instanciada) se existe um morfismo (vamos chamá-lo de μ) ligando $\mathcal{F}_i$ e $\mathcal{F}'_i$ que faça comutar



Vamos dizer que dois programas computam a mesma função se e somente se, para a mesma entrada, produzem a mesma saída. É claro que estamos, neste momento, nos referindo apenas a programas que páram. Considere, então, um determinado atributo ϕ, representando o arquivo de saída, e uma seção S^ϕ, que descreve sua evolução em $\mathcal{T}^{op}$. Existe um aberto Y em $\mathcal{T}^{op}$ que é ideal, no sentido em que permite visualizar toda evolução de todos os atributos até o momento da parada, ou seja, até o momento em que todos os atributos são mapeados no conjunto vazio. Vamos dizer que duas implementações $\mathcal{F}$ e $\mathcal{F}'$, instanciadas para a mesma entrada, computam a mesma função se e somente se $S^\phi(Y) = S'^\phi(Y)$, onde ϕ é o atributo 'arquivo de saída' e S e S' são seções de $\mathcal{F}$ e$\mathcal{F}'$. Podemos, então, estabelecer que, para o caso de programas que páram, i.e., em que se pode identificar o aberto ideal Y, o morfismo $\mu : \mathcal{F} \to \mathcal{F}'$ é determinado pela equivalência acima.

No caso de programas que não páram, não se pode dizer que dois programas computam a mesma função se e somente se, para a mesma entrada, produzem a mesma saída. Mesmo em casos patológicos, como um programa que efetua a soma e pára, e outro que efetua a soma, e entra em *loop*, não se pode assumir que a saída do segundo é o resultado da soma. A soma é um resultado parcial, já que não houve terminação.

```
input(a,b)          input(a,b)
c := a+b            c := a+b
output(c)           output(c)
                    loop
                    end loop
```

Por outro lado, assumir que programas em *loop* computam $\perp$, e extrair daí o morfismo μ, não é uma estratégia interessante a ser adotada aqui, visto que identifica, numa única classe, programas que podem ser radicalmente diferentes. Imagine, por exemplo, um sistema operacional, que roda incessantemente enquanto a máquina está ligada, efetuando trabalhos úteis, e um *loop* como o descrito acima. Gostraríamos de caracterizar que estes programas são radicalmente diferentes um do outro.

A direção oposta a de unificar todos os *loops* seria admitir que não se pode determinar semelhanças entre *loops*, e portanto, não existe um morfismo μ conectando-os. Esta estratégia, por um lado diferencia os *loops*, mas, por outro, deixa de captar características que podem ser interessantes, como, por exemplo, o fato de que os dois programas interativos abaixo recebem dois valores para somar, efetuam a soma, e mostram o resultado.

```
loop                   loop
   input(a,b)             output("digite dois valores")
   c := a+b               input(c,d)
   output(c)              e := c+d
end loop                  output("a soma e'")
                          output(e)
                       end loop
```

O meio termo para essas abordagens, seria, então, caracterizar a semelhança com relação a certas características.

Já que se tratam de *loops*, devemos observar que os atributos se comportam de maneira uniforme: uma variação no valor de um atributo é conseqüência da execução de um mesmo comando do *loop*, e produzirá o mesmo efeito, a menos de mudanças em certos parâmetros. Assim, teremos que a seção que descreve um atributo em um *loop* terá um desenho monótono (no sentido usual da palavra). Daí, podemos identificar semelhanças nos *loops* a partir destas formas.

Vamos, então, dizer que dois *loops* $\mathcal{F}$ e $\mathcal{F}'$, têm um comportamento similar em relação a uma classe de seções C_S se e somente se existe uma classe de seções C'_S em $\mathcal{F}'$ tal que $C'_S \equiv C_S$, onde C_S é uma classe de seções de $\mathcal{F}'$. O sinal $\equiv$ está indicando que, para toda seção em uma das classes, pode-se encontrar uma seção na outra classe, que equivale a primeira, a menos de alguma deformação.

Finalmente, definimos μ.

O morfismo $\mu : \mathcal{F} \rightarrow \mathcal{F}'$ existe sse

(i) Existem os abertos Y e Y', ideais em $\mathcal{T}^{op}$, e as seções S e S', tal que $S^{\phi}(Y) = S'^{\phi}(Y')$, onde ϕ é o atributo 'arquivo de saída', ou

(ii) Existem seções S e S', tal que $S = S'$ a menos de uma deformação.

9 Em direção à paradigmas

Nossa abordagem conta com a particularidade de ser construída em diversos níveis de abstração, e de preservar a geometria na qual se baseia a nossa intuição de funções e programas. Isto permite avaliar as conseqüências de um determinado fato em diversos níveis de abstração. Com iso, pretendemos reduzir implementações ao que há de mais básico em qualquer paradigma: as evoluções no espaço-tempo. Utilizando a visão geométrica, que permite identificar os resultados com mais facilidade, pretendemos alcançar um meio de distinguir e agrupar adequadamente cada *trace*, e verificar suas conseqüências em níveis mais abstratos.

O passo seguinte a este trabalho consiste, então, em realizar experimentos com programas de diferentes paradigmas. Pretendemos definir um conjunto de atributos que inclua características inerentes a cada paradigma, e avaliar, em termos da geometria das seções, as semelhanças e diferenças com relação às diferentes abordagens.

10 Conclusão

A visão geométrica de funções computáveis é o princípio de uma série de investigações a respeito de implementações. Estas investigações têm, no entanto, um aspecto diferente: elas são efetuadas em um ambiente estratificado, ou seja, com diferentes níveis de abstração. Com isso, no nível de implementações podemos estudar *traces* e seus relacionamentos, enquanto que no nível de funções computáveis pode-se estudar implementações e seus relacionamentos.

Esta abordagem vem de encontro à dificuldade encontrada em caracterizar o conceito de paradigmas de linguagem de programação. A razão desta dificuldade está no fato de que os métodos formais conhecidos já incorporam certos aspectos que fazem parte do paradigma, conforme ressaltamos na introdução deste trabalho. Isto ocorre porque a análise é feita sob um único ponto de vista, utilizando um ferramental muito próximo das próprias linguagens de programação. Sendo assim, propomos um ambiente que permite que o estudo se focalize ora no nível da linguagem de programação (implementações), ora no nível das execuções (*traces*), ora no nível das funções computáveis. Nosso ambiente provê meios para passar de um nível a outro, o que permite deduzir, em todo os níveis, as conseqüências obtidas num determinado nível. De uma forma ou de outra, as entidades do 'mundo da programação' podem ser representadas geometricamente, o que facilita a visualização e obtenção de resultados não necessariamente intuitivos.

References

[1] J. Goguen. *Sheaf Semantics for Concurrent Interacting Objects.* CSLI, Springer, 1991.

[2] Robert Goldblat. *Topoi - The Categorial Analysis of Logic.* North Holland Publishing Company, Holland, 1979.

[3] A Sernadas H. Ehrich, J. Goguen. *A categorical theory of objects as observed processes.* In Proceedings, REX/FOOL Workshop on Foundations of Object Oriented Languages, Springer, 1991.

[4] F. G. Pagan. *Formal Specification of Programming Languages.* Prentice/Hall International, Englewood Cliffs, 1981.

[5] David A. Schmidt. *Denotational Semantics: a methodology for language development.* Wm C. Browm, Dubuque, Iowa, 1988.

[6] R. F. C. Walters. *Categories and Computer Science.* University Press, Cambirdge, 1991.