GRAFOS, ÁRVORES E TERMOS EM FOL

RAUL H.C.LOPES

1. Um problema interessante

A figura 1 apresenta um grafo dirigido: um conjunto de vértices e arcos conectando pares de vértices. Um grafo dirigido pode representar vias de trânsito em uma cidade ou nós em uma estrutura de dados de um programa. Questões interessantes que podem ser resolvidas com a ajuda de um grafo são, por exemplo: existe um caminho (sempre seguindo a direção dos arcos) que leve de de um vértice x a um vértice y.

Mais formalmente:

Definição 1. *Um grafo dirigido é um par* (V, A)*, onde:*

- V *é um conunto de vértices;*
- $A \subseteq V \times V$*, A sendo chamado conjunto de* arcos *do grafo.*

O conjunto de arcos do grafo denota uma relação binária não simétrica entre os vértices do.

Dado um arco (x, y):

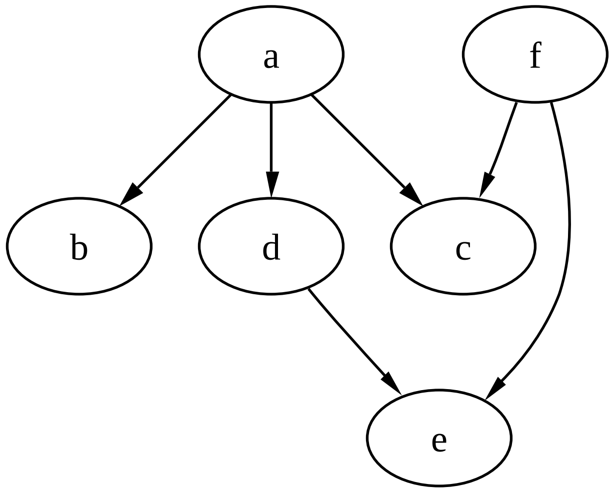

Figura 1. Grafo dirigido

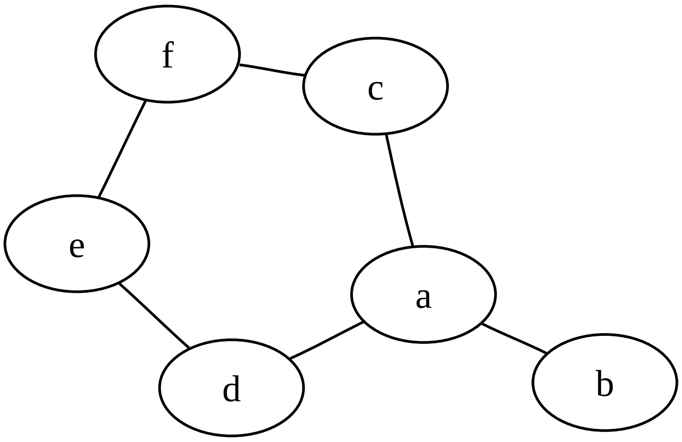

FIGURA 2. Um grafo não dirigido

- x é chamado vértice *inicial*;
- y é vértice *final*;
- x e y são adjacentes;

2. Grafos e Árvores

O termo grafo, em geral, refere-se a uma estrutura contendo vértices e pares não ordenados de vértices, indicando ligações não direcionadas entre pontos.

Definição 2. *Um grafo é um par* (V, E)*, onde:*

- V *é um conjunto de vértices.*
- $E \in \{\{x, y\} | x \in V, y \in V\}$*,* E *sendo chamado conjunto de* arestas *do grafo.*

O conjunto de arestas do grafo denota uma relação binária simétrica entre os vérticesdo mesmo.

A figura 2 representa uma versão não dirigida do grafo da figura 1.

Note que, neste texto, o termo *grafo* refere-se a um grafo não dirigido.

Definição 3. *Um vértice* x *é incidente a uma aresta* e *se* $x \in e$*. Se* $x, y \in e$*, então* x *e* y *são adjacentes. Uma aresta* $\{x, y\}$ *poderá ser apresentada simplesmente como* xy*, ou* yx*, e* X–Y *denotará o conjunto de todas as arestas com vértices em* X *e* Y*.*

Definição 4. *Uma seqüência de vértices* e_0, e_1, e_{n-1} *de um grafo* (V, E)*, onde cada par* $\{e_i, e_{i+1}\}$ *é uma aresta em* E *é chamado de trajeto. Um trajeto que não repete arestas é chamado tour. Um trajeto que não repete vértices é um caminho. Uma caminho que repete o primeiro e último vértices 'e chamada de ciclo.*

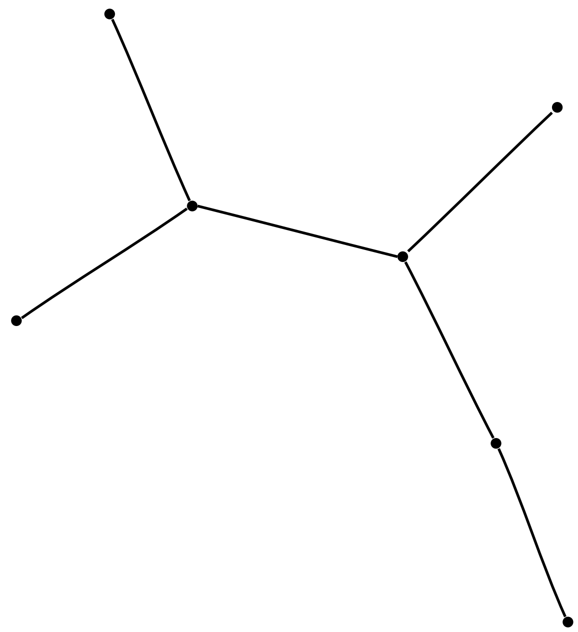

Figura 3. Uma árvore

Note que os termos acima não estão padronizados na literatura sobre grafos. Veja por exemplo, [4], [2] e [3], onde o termo *path* é usado em três sentidos diferentes.

3. Conectividade e Árvores

Definição 5. *Um grafo (V, E) é dito conectado quando para qualauqer par de vértices $x, y \in V$ existe um trajeto de x a y em (V, E).*

Definição 6. *Uma árvore é um grafo conectado sem ciclos.*

A figura 3 apresenta um exemplo de árvore.

Definição 7. *Árvore geradora de um grafo (V, E) é um subgrafo de (V, E) que contém todos os vértices de V e é uma árvore.*

Frequentemente, em estudos em *Estruturas de Dados* árvores são apresentadas com um vértice distinto do qual *descendem* todos os outros vértices, como na figura 4, ou na figura 5, que apresenta a árvore como um grafo dirigido.

Definição 8. *Um vértice x de uma árvore (V, E) é chamado folha (ou nó externo) quando só existe um par $xy \in E$. Outros nós (ou vértices) da árvore são chamados internos.*

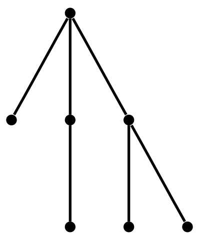

FIGURA 4. Routed tree

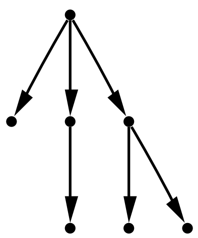

FIGURA 5. Árvore dirigida

4. Problemas interessantes

Alguns problemas interessantes que podem ser resolvidos com grafos são:

- Dados vértices a e b, existe um trajeto entre a e b?
- Dado vértice a, encontre todos os y tal que existe trajeto de a para y.
- Qual é o caminho mais curto entre x e y?
- Qual é o caminho mais curto entre um vértice x e todos os outros vértices do grafo.
- Qual é o maior caminho entre a raiz de uma árvore e qualquer outro vértice.

5. Solução usando lógica

A solução para o primeiro desses problemas poderia facilmente ser obtida usando uma formalização em lógica de primeira ordem para representar o grafo em questão e aquelas de suas propriedades que dizem respeito ao problema em estudo. Tal formalização deveria conter axiomas que descrevam as arestas do grafo e axiomas que descrevam uma lei de transitititividade para a relação de arestas. Por exemplo, o fato de que ab é aresta do grafo em estudo poderia ser representado pelo axioma:

$$edge(a, b)$$

A reflexividade da relação binária de arestas de um grafo não dirigido seria explicitada através do axioma

$$adjacent(x, y) \Rightarrow adjacent(y, x)$$

Finalmente, a existência de um trajeto entre dois vértices quaisquer seria axiomatizada através de uma lei de transitividade para a relação de arestas.

$$\forall x : \forall y : \forall z : adjacent(x, y) \wedge walk(y, z) \Rightarrow walk(x, z)$$

A partir dessa formalização a primeira questão, *se existe um trajeto* entre dois vértices a e b, seria resolvida pela prova (ou não) do teorema:

$$walk(a, b)$$

A segunda questão, que busca todos os y tal que *existe um trajeto de a a y*, poderia ser resolvida pela prova (possivelmente automática) do seguinte teorema, com a construção dos devidos valores de y que o satisfazem:

$$\exists y : walk(a, y)$$

6. Árvores para representar tipos indutivos

Árvores podem ser usadas para representar tipos indutivamente definidos. Por exemplo, expressões são exemplos clássicos de tipos indutivamente definidos. Expressões aritméticas podem ser vistas como pertencentes a uma classe indutiva cujos elementos básicos da classe são números inteiros. Outros elementos da classe são obtidos pela aplicação de um construtor (operador aritmético) a uma ou mais expressões aritméticas. Por exemplo, a expressão $2 * (3 - 4)$ resulta da aplicaçãodo operador $*$ às expressões 2 e $(3 - 4)$, a qual é definido por um processo recursivo. Ela pode ser representada pela árvore da figura 6. Note que expressões simples, compostas apenas de números, aparecem como folhas.

```
data Ae =  Num Int
         | Mult Ae Ae
         | Sub  Ae Ae
```

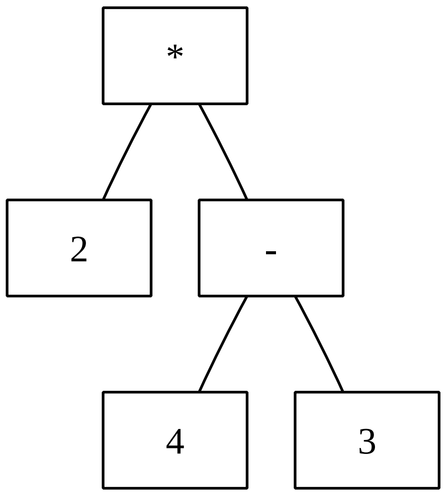

FIGURA 6. Árvore para $2 * (3 - 4)$

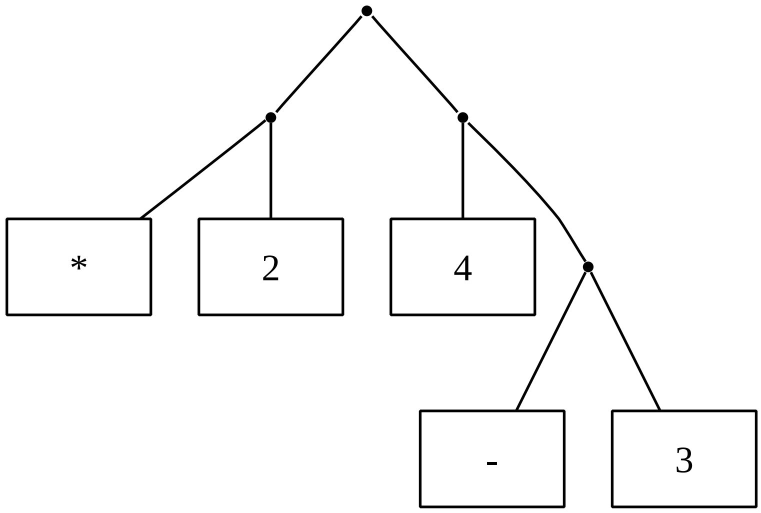

FIGURA 7. Leaf tree para $2 * (3 - 4)$

Uma apresentação alternativa para a mesma expressão é dada pela figura 7, que usa uma *leaf tree* para apresentar a interpretação de Curry dos operadores de adição e multiplicação: $*2x$ é interpretado como uma função que mapeia o inteiro 2 para uma função que, dado x, retorna o seu dobro.

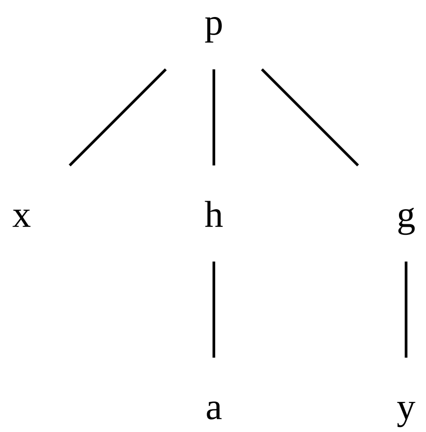

FIGURA 8. Árvore para $p(x, h(a), g(y))$

Um tipo de dados em Haskell para representar interpretação poderia ser:

```
data CurriedAe =   Appl Curr  CurriedAe
                 | Num Int

data Curr =   Mult Int
            | Sub Int
```

7. Fórmulas como árvores

Fórmulas podem ser apresentadas como árvores. Veja, por exemplo, o predicado $p(x, h(a), g(y))$, que é representado como árvore na figura 8.

Esse predicado poderia ser representado em Haskell, usando o tipo *Atom* para representar um predicado aplicado a uma lista de termos. O tipo *Term* seria usado para seus argumentos.

```
data Atom =   Pred String [Term]


data Term =   Appl String [Term]
            | Var  String

t0 = Var "x"
t1 = Appl "h" [Appl "a" []]]
t2 = Appl "g" [Var "y"]
v = Pred "p" [t0, t1,t2]
```

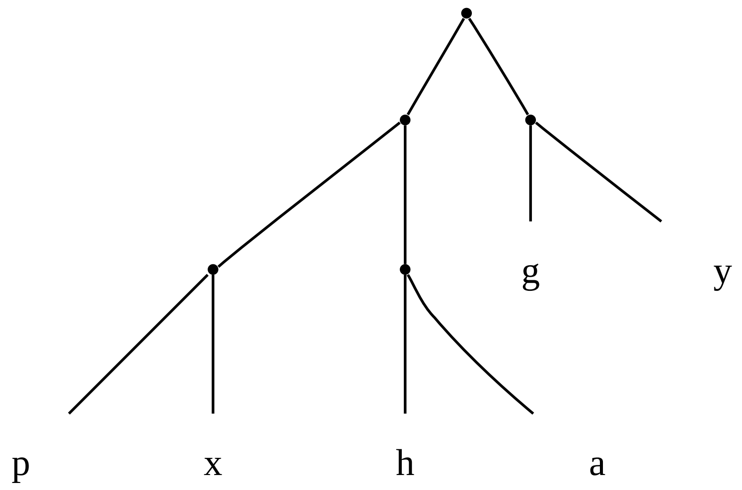

FIGURA 9. Leaf tree $p(x, h(a), g(y))$

Una repressentação alternativa para esse mesmo aparece na figura 9, que pode ser vista como a árvore de $(((px)(ha))(gy))$.

7.1. **Operadores de um termo.** No final, um termo é uma variável ou um operador aplicado a uma lista de termos, que pode ser visto como uma função que, aplicada a uma lista de argumentos, constói um objeto da teoria em estudo. Um exercício interessante consiste em coletar todos os operadores e variáveis que ocorrem em um termo. Esse problema na verdade consiste em um exercício de caminhamento em grafos. Uma primeira alternativa consiste em executar um caminhamento na largura da árvore: os operadores que estão no mesmo *nível* sendo coletados primeiro, como no algoritmo **bfscollect**, abaixo.

```
bfscollect [] = []
bfscollect (x:xs) = [(ftor x)]++(bfscollect (xs++(fargs x)))

ftor (Var x) = x
ftor (Appl x _) = x

fargs  (Var x) = []
fargs (Appl _ xs) = xs
```

Note que o algoritmo acima coleta os operadores de uma lista de termos. Uma alternativa ao caminhamento em largura é o caminhamento em profundidade, apresentado na função **dfscollect**, abaixo.

```
dfscollect [] = []
```

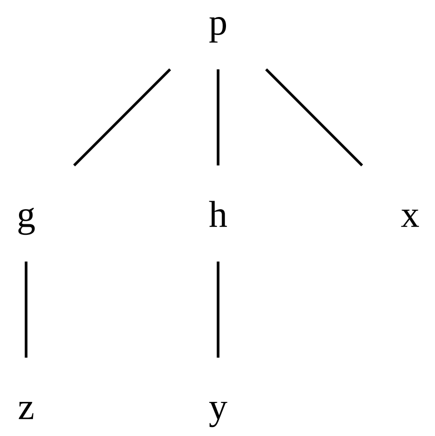

FIGURA 10. Árvore para $p(g(z), h(y), x)$

```
dfscollect (x:xs) = [(ftor x)]++(dfscollect ((fargs x)++xs))
```

8. Unificadores

Um processo importante em prova automática de teoremas é aquele em que valores (instâncias) são encontrados para variáveis, permitindo encontrar dois termos comuns de duas instâncias diferentes: por exemplo, permitindo descobrir que $p(x, h(a), g(z)$ da figura 8 tem uma instância em comoum com $p(g(z), h(y), x)$, da figura 10, que consiste em $p(g(a), h(a), g(a))$, representado na figura 17.

8.1. **Substituições.**

Definição 9. ***Conjunto substituição.*** *Uma substituição é um par mapeando uma variável para um termo. Um conjunto de pares mapeando variáveis para termos é chamado conjunto subsituição.*

O conjunto substituição que mapeia $p(x, h(a), g(z)$ para $p(g(z), h(y), x)$ consiste no seguinte conjunto de pares:

$$\{(x, g(a)), (y, a), (z, a)\}$$

A aplicação de uma substituição (x, t) a um termo u consiste em substituir em u simultaneamente todas as ocorrências de x por t. Esse processo é facilmente descrito por recursão sobre a estrutura indutiva do termo, como no algoritmo abaixo, onde **varSubst** substitui um variável por um termo em um termo dado. Já a função **subst**, também apresentada a seguir, aplica um conjunto substituição a um termo dado.

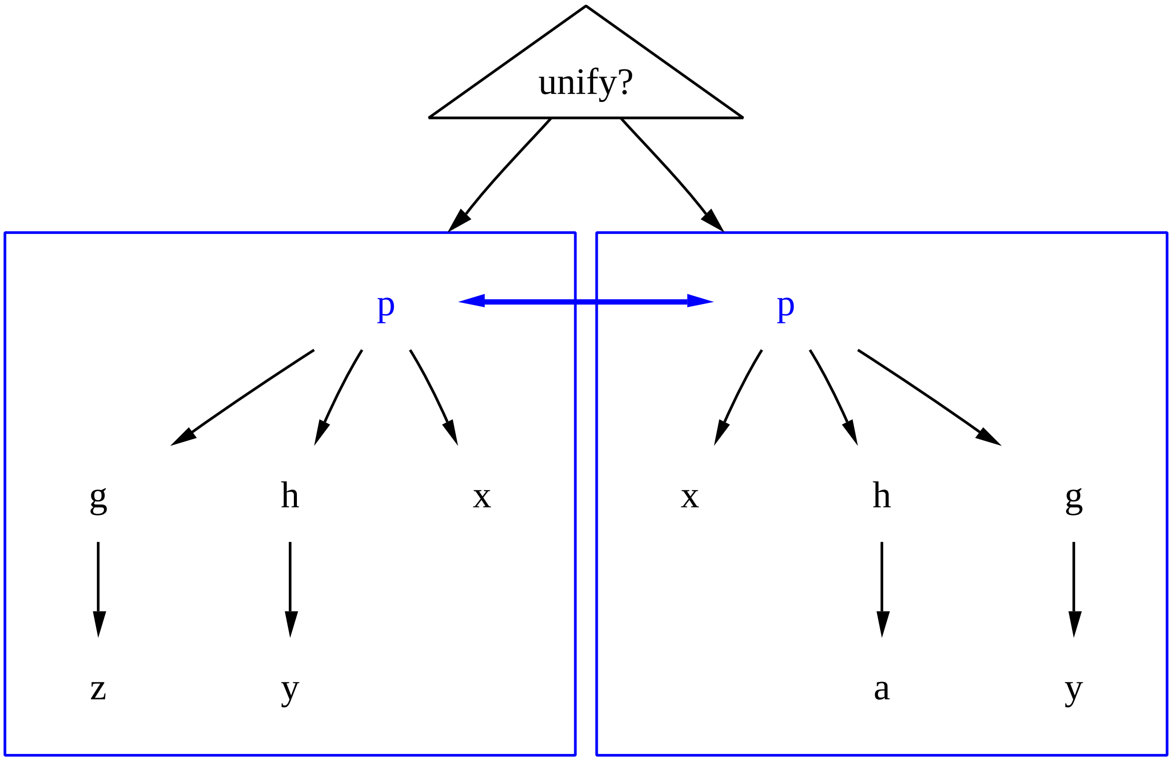

Figura 11. Unificação de $p(x, h(a), g(y))$ e $p(g(z), h(y), x)$

```
varSubst ((Var y),t) (Var x)=
   if  x == y then t
   else (Var x)
varSubst sp (Appl f xs) = Appl f (map (varSubst sp) xs)

subst xs t = foldr subst t xs
```

8.2. **Unificação.** Um unificador para dois termos s e t é um mapeamento θ de variáveis para termos, conjunto substituição, que quando aplicado a s e t produz dois termos iguais: $\theta.s = \theta.t$. Um processo para encontrar o unificador mais geral entre dois termos consite na resolução de um sistema de equações que tem como equação inicial única $s = t$.

Por exemplo, a unificação de $p(x, h(a), g(y))$, figura 8, e $p(g(z), h(y), x)$, figura 10, inicia com o sistema que tem uma única equação, dada pelo par

$$\{(p(x, h(a), g(y)), p(g(z), h(y), x))\}$$

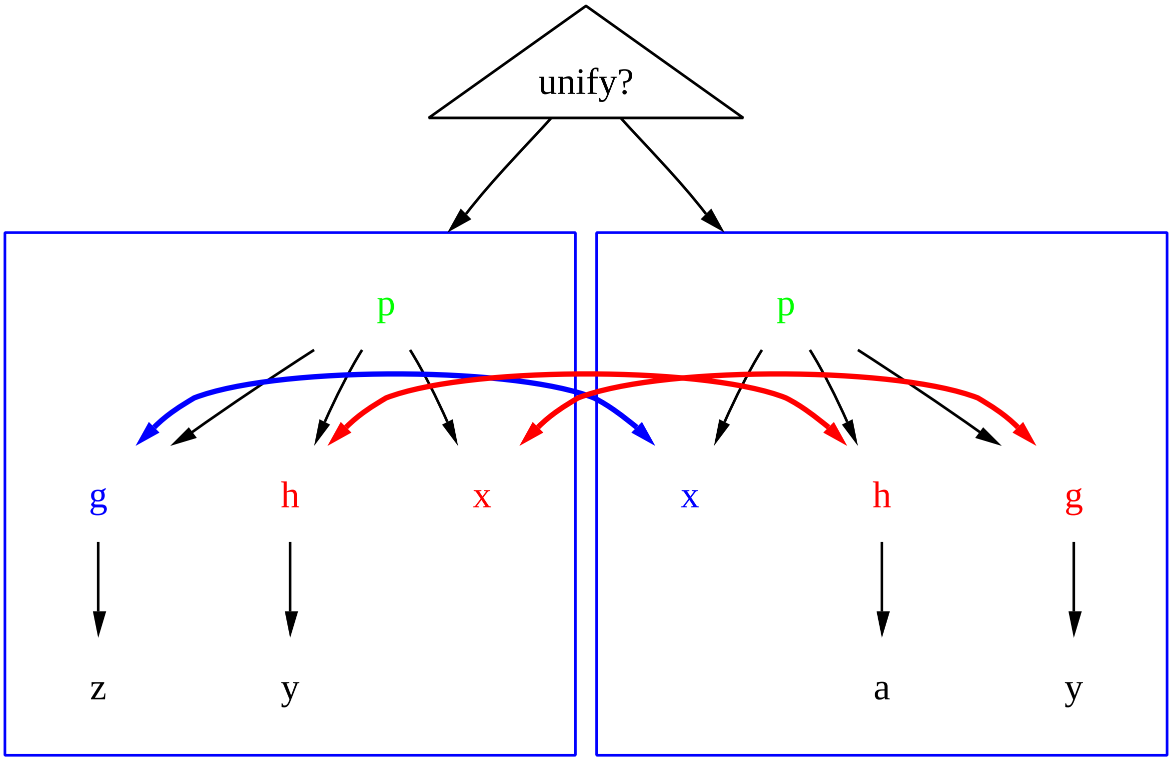

FIGURA 12. Par $(x, g(z))$

A figura 11 mostra essa equação, marcando em cor azul o primeiro par de subtermos a considerar.

A resolução dessa equação consiste em substitui-la por três novas equações dadas pelos pares

$$\{(x, g(z)), (h(a), h(y)), (g(y), z)\}$$

O passo de reescrita desse par inicial por um conjunto de pares, figura 12, definindo novas equações no sistema, pode ser descrito como:

Definição 10. ***Unificação de par função-função.***

Dado um sistema

$$\{(f(u_0, u_1, \ldots, u_{m_f}), g(v_0, v_1, \ldots, v_{m_g})), (s_1, t_1), \ldots, (s_{n_1}, t_{n_1})\}$$

substitua $(f(u_0, u_1, \ldots, u_{m_f}), g(v_0, v_1, \ldots, v_{m_g}))$ *por* $\{(u_0, v_0), \ldots, (u_{m_f}, v_{m_g})\}$, *se* $f = g$. *Caso* $f \neq g$, *o sistema não é unificável.*

A cor azul na figura 12 destaca o primeiro par a considerar. A cor verde destaca vértices que já foram considerados, fechados. A cor vermelha marca que vértices que já foram encontrados em algum par, mas

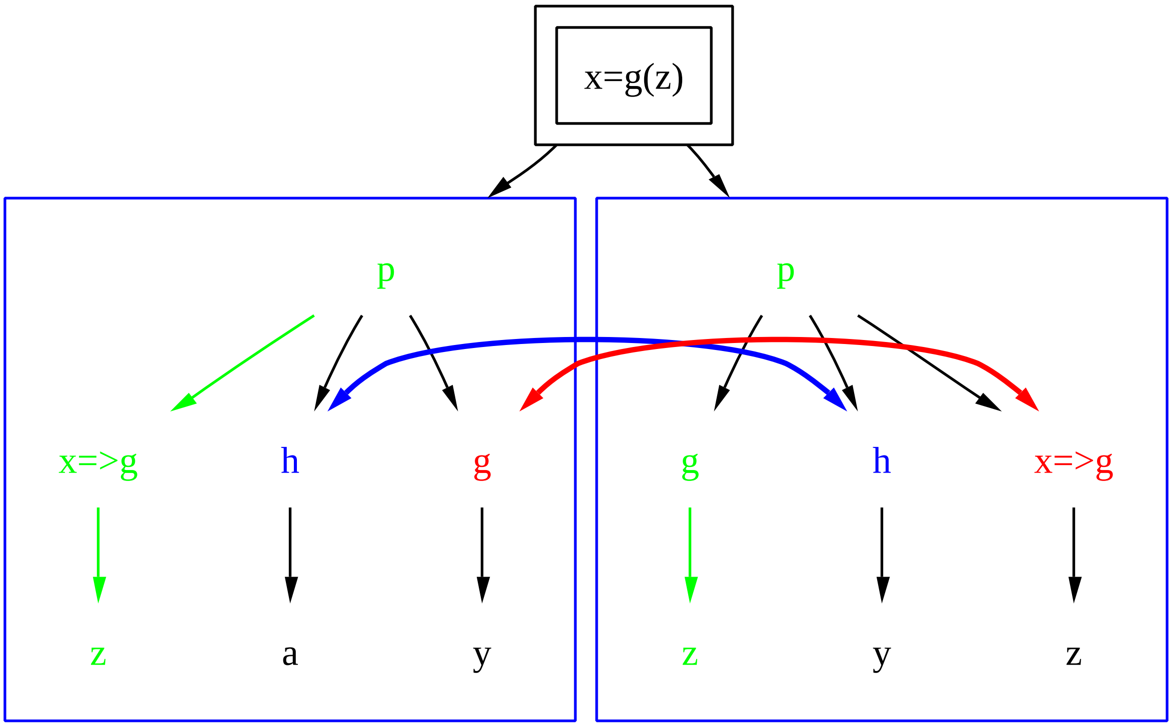

Figura 13. Par $(h(a), h(y))$

não foram explorados. A cor preta marca vértices não explorados. Note que a escolha $(x, g(z))$ como primeiro par a unificar é arbitrária: qualquer um dos outros pares marcados em vermelho poderia ter sido selecionado. Na verdade, essa escolha reflete um estratégia de exploração em largura das árvores de termos.

Assumindo que x, y e z são variáveis, x é mapeado para $g(z)$, constituindo a primeira substituição, anotada no topo da figura 13, alterando o sistema de equações a resolver. Essa passo de reescrita de equações do sistema pode ser descrito como:

Definição 11. ***Unificação de par variável-termo.*** *Dado um sistema*

$$\{(s_0, t_0), (s_1, t_1), \ldots, (s_{n_1}, t_{n_1})\}$$

onde s_0 *(*t_0*) é variável e* s_0 *(*t_0*) não ocorre em* t_0 *(*s_0*), e assumindo* θ *como unificador parcial, considere como novo sistema de equações*

$$\theta'.\{(s_1, t_1), \ldots, (s_{n_1}, t_{n_1})\}$$

onde

$$\theta' = \{(s_0, t_0)\}$$

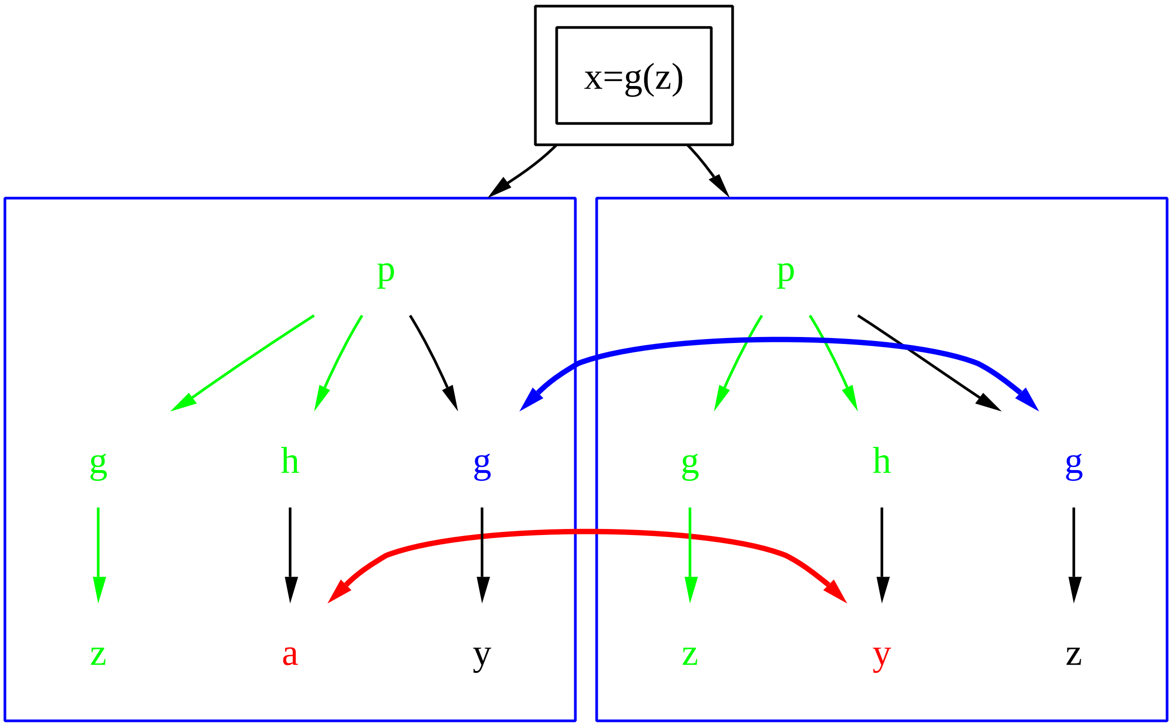

Figura 14. Par $(g(y), g(z))$

e assuma $\theta'.\theta$ como novo unificador parcial

Definição 12. ***Composição de unificadores*** *Dados unificadores*

$$\theta = \{(x_0, t_0), \ldots, (x_{n-1}, t_{n-1})\}$$

e σ a composição de σ e θ é definida como:

$$\sigma.\theta = \{(x_0, \sigma.t_0), \ldots, (x_{n-1}, \sigma.t_{n-1})\} \cup \sigma$$

A figura 13 coresponde à situação:

$$\{(h(a), h(y)), (g(y), g(z))\}, \theta = \{(x, g(z))\}$$

onde θ é o unificador identificado até o momento.

No próximo passo, o par $(g(y), g(z))$ é considerado, como mostra a figura 14. Dado que eles apresentam o mesmo operador de função, esse par será substituído por nova equação: (y, z).

A figura 15 mostra o próximo par a considerar: (a, y), em azul. O par é trivialmente unificável, uma vez que y é variável e a não contém ocorrência de y.

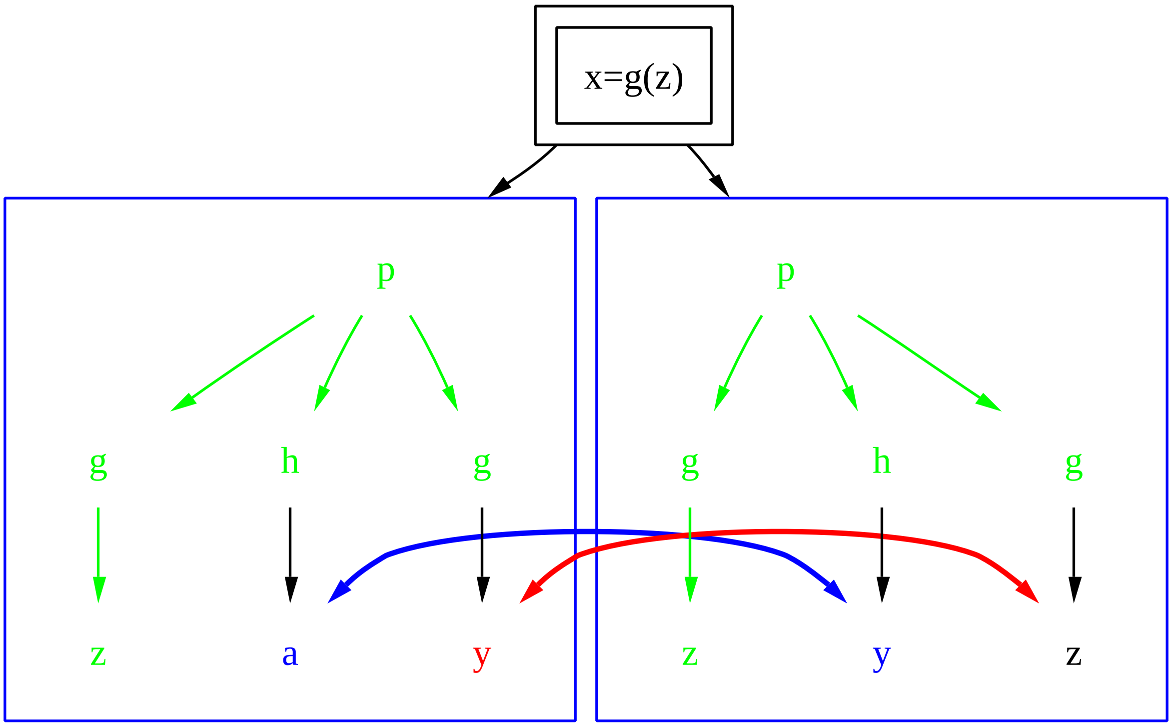

FIGURA 15. Par (a, y)

A figura 16 mostra a situação em que o unificador parcial é

$$\theta = \{(x, g(z)), (y, a)\}$$

e o sistema de equações contém uma única equação:

$$\{(a, z)\}$$

A árvores da figura são resultantes de aplicar θ aos termos iniciais.

Na figura 17, os dois termos estão unificados e o mapeamento θ define o unificador dos termos iniciais s e t:

$$\theta = \{(x, g(a)), (y, a), (z, a)\}$$

Observe que θ é a composição de seu valor no passo anterior

$$\theta = \{(x, g(z)), (y, a)\}$$

com a substituição obtida no último passo

$$(z, a)$$

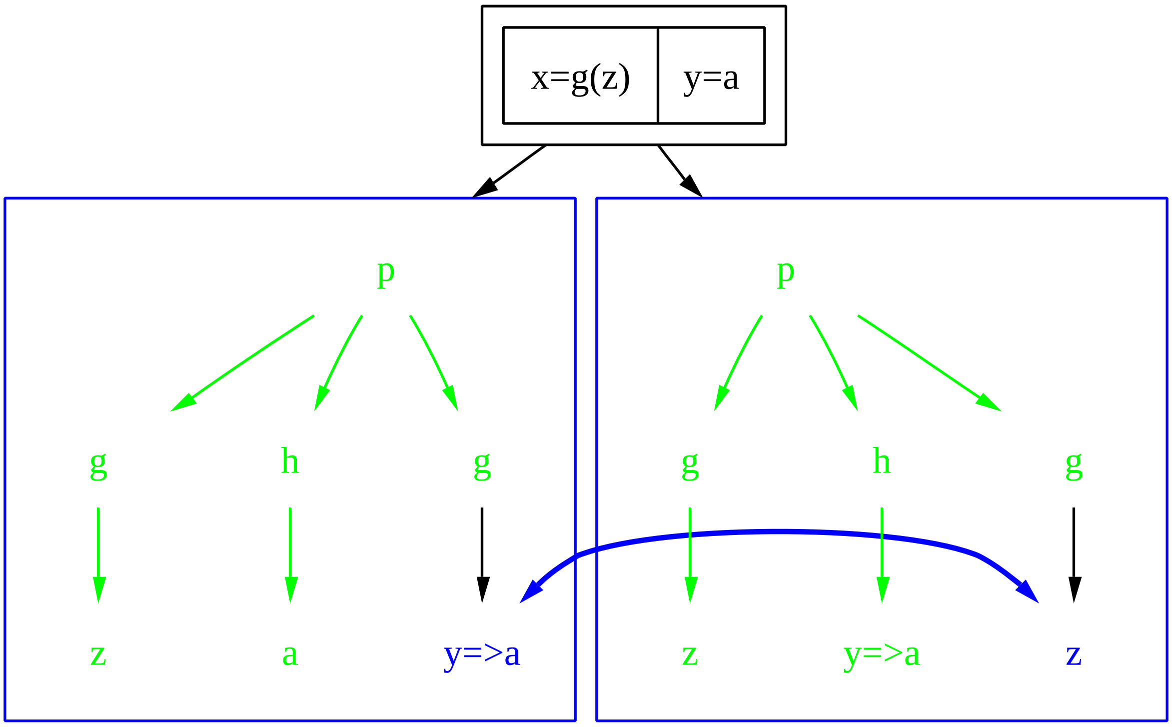

Figura 16. Par (a, z)

9. Principio da resolucao

Uma das motivações iniciais deste rascunho consistia em representar logicamente as propriedades de um grafo responder a questões sobre o mesmo usando prova de teoremas.

Um grafo pode ser facilmente representado em lógica por um conjunto de axiomas que estabelecem:

- As arestas do grafo e, possivelmente, seus pesos. Por exemplo,

$$ab$$

  definindo que existe aresta conectando a e b.
- Em grafos não dirigidos, a simetria da relação de arestas:

$$\forall x : \forall y : xy \Rightarrow xedgeyx$$

- a reflexividade de relação que define acessibilidade. Por exemplo,

$$\forall x.a(x, x)$$

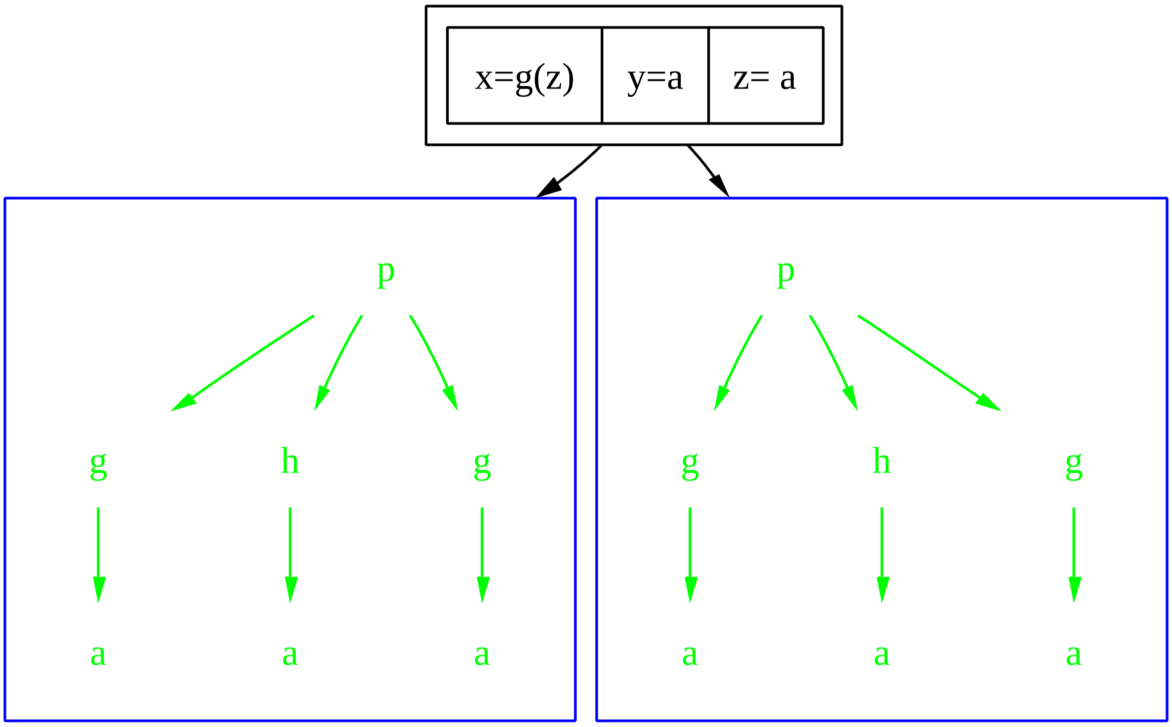

FIGURA 17. Termos Unificados

- a acessibilidade de um vértice a outro através de uma aresta:

$$\forall x : \forall y : xy \Rightarrow a(x, y)$$

- A transitividade da relação de acessibilidade:

$$\forall x : \forall y : \forall z : a(x, y) \wedge a(y, z) \Rightarrow a(x, z)$$

9.1. **Representação clausal.** Assumindo o contexto de lógica clássica, qualquer sentença da lógica pode ser representada usando apenas as quatro constantes lógica para quantificação uniuversal, conjunção, disjunção e negação. Uma representação usual é a da notação clausal: axiomas são representados como disjunções de literais, átomos ou negaçãode átomos, e teorias são apresentadas como conjunções de cláusulas. De forma geral, fórmulas e teoremas são apresentadas como conjunções de cláusulas, de agora em diante apresentadas como conjuntos de cláusulas.

9.2. **Prova por contradição.** A processo de prova de teoremas quando representação clausal está envolvida é frequentemente baseado no **Princípio da Resolução**, veja por exemplo [1], que consiste essencialmente em um método de prova por contradição. A busca da prova de um teorema segue os seguintes passos:

(1) representação da teoria como conjunção de cláusulas;
(2) negação do teorema e sua conversão notação clausal;
(3) busca da contradição pela aplicação repetida de algumas regras de inferência básicas como: resolução binária, fatoração, eliminação de tautologias e cláusulas subjugadas.

Definição 13. ***Resolução binária.*** *Dadas duas cláusulas*

$$C = \{C_0, \ldots, C_{n-1}\} \text{ e}$$
$$D = \{D_0, \ldots, D_{m-1}\}$$

tais que:

- *C e D não têm variáveis em comum;*
- *$C_i \in C$;*
- *$\neg D_j \in D$;*
- *existe um unificador θ de C_i e D_j, $\theta.C_i = \theta.D_j$;*

O resolvente binário de C e D é a cláusula:

$$\theta.(C - \{C_i\}) \cup \theta.(D - \{D_j\})$$

Definição 14. ***Fatoração binária.*** *Dada uma cláusula*

$$C = \{C_0, \ldots, C_{n-1}\} \text{ e}$$

tal que:

- *$C_i \in C$;*
- *$C_j \in C$;*
- *existe um unificador θ de C_i e C_j, $\theta.C_i = \theta.C_j$;*

É fator binário de C a cláusula:

$$\theta.(C - \{C_j\})$$

Definição 15. ***Subjugação de cláusulas.*** *Uma cláusula$D$ é subjugada por uma cláusula C se: existe uma substituição θ que não substitui variáveis de D tal que*

$$\theta.C \subseteq D$$

Definição 16. ***Tautologia.*** *Uma cláusula é uma tautologia se ela contém literais L e M tais que L é a negação de M.*

9.3. **Busca da prova.** A busca de prova em um provador baseado em resolução consiste na aplicação repetiva de variantes da **resolução binária** e **fatoração binŕia** até a geração da contradição (cláusula vazia) ou até o ponto fixo ser atingido. Geralmente, estratégias *de poda* são usadas para tentar conrolar a explosão do espaço de pesquisa. Duas estratégias comuns consistem na eliminação de tautologias e cláusulas subjugadas.

9.4. **Programação em lógica.** Programação em lógica consiste idealmente[1] na codificação de programas como axiomas em lógica de primeira ou mais alta ordem. Nesse campo, uma alternativa[2] frequentemente encontrada consiste em codificar os programas como conjuntos de cláusulas. Uma execução do programa consiste na prova de um teorema, este representando uma consulta à base de axiomas que é o programa. Consultas frequentemente envolvem variáveis existencialmente quantificadas, como, por exemplo, em:

$$\exists x : a(a, x)$$

que, no exemplo de acessibilidade em grafos anteriormente apresentado, codifica a questão: "Existe um vértice x tal que exista um trajeto de a a x?" Se o provador envolvido for "esperto", ele pode responder à questão, em caso positivo, construindo todos os valores de x que que a satisfazem.

9.5. **Trabalho Prático 1.** Será que você consegue implementar um provador para a lógica de primeira ordem? E ambiente de programação em lógica?

Referências

[1] C. Chang and R.C. Lee, *symboli logic and mechanical theorem proving*, Academic Press, 1973.
[2] Reinhard Diestel, *Graph theory*, Springer-Verlag, 2000.
[3] Michel Gondran, Michel Minoux, and Steven Vajda, *Graphs and algorithms*, John Wiley & Sons, 1984.
[4] W. T.Tutte, *Connectivity in graphs*, University of Toronto Press, 1966.

[1]Apenas idealmente! Porque ao menos no mundo da linguagem Prolog, que veio a se tornar o sinônimo infeliz de programação em lógica, um programa em "lógica" tem freqüentemente tantas linhas de *hackering* e caracteristicas extra-lógicas quanto tem de axiomas em lógica de primeira ordem.

[2]Definitivamente não a única: existem sistemas programação em lógica baseados em fragmentos de lógica de alta ordem, lógica construtivista, lógica linear, etc.