PICAT: Uma Linguagem de Programação Multiparadigma

Miguel Alfredo Nunes, Jeferson L. R. Souza, Claudio Cesar de Sá

miguel.nunes@edu.udesc.br
jeferson.souza@udesc.br
claudio.sa@udesc.br

Departamento de Ciência da Computação
Centro de Ciências e Tecnológias
Universidade do Estado de Santa Catarina

16 de abril de 2019

Contribuições

- Alexandre Gonçalves;
- João Herique Faes Battisti;
- Paulo Victor de Aguiar;
- Rogério Eduardo da Silva;
- Hakan Kjellerstrand – (`http://www.hakank.org/picat/`)
- Neng-Fa Zhou – (`http://www.picat-lang.org/`)
- Outros anônimos que auxiliaram na produção deste documento;

Tipos de Dados e Variáveis – Introdução – I

- Em projetos de linguagens de programação há dois tipos verificação do tipo de dados: **estática** e **dinâmica**

Tipos de Dados e Variáveis – Introdução – I

- Em projetos de linguagens de programação há dois tipos verificação do tipo de dados: **estática** e **dinâmica**
- A verificação de tipos dados estática em *tempo de compilação*.

Tipos de Dados e Variáveis – Introdução – I

- Em projetos de linguagens de programação há dois tipos verificação do tipo de dados: **estática** e **dinâmica**
- A verificação de tipos dados estática em *tempo de compilação*.
- Enquanto a dinâmica em *tempo de execução*.

Tipos de Dados e Variáveis – Introdução – I

- Em projetos de linguagens de programação há dois tipos verificação do tipo de dados: **estática** e **dinâmica**
- A verificação de tipos dados estática em *tempo de compilação*.
- Enquanto a dinâmica em *tempo de execução*.
- Linguagens fortemente tipadas, tais como C, Java e Pascal, exigem que o tipo do dado (conteudo) seja do mesmo tipo da variável ao qual este valor será atribuído. Tudo isto é pré-definido durante a fase da *compilação*.

Tipos de Dados e Variáveis – Introdução – II

- Nas linguagens interpretadas, com uma máquina virtual, esta definição é feita durante a *execução* do programa

Tipos de Dados e Variáveis – Introdução – II

- Nas linguagens interpretadas, com uma máquina virtual, esta definição é feita durante a *execução* do programa
- Prós e contras para o que é melhor, a discussão fica de lado neste momento

Tipos de Dados e Variáveis – Introdução – II

- Nas linguagens interpretadas, com uma máquina virtual, esta definição é feita durante a *execução* do programa
- Prós e contras para o que é melhor, a discussão fica de lado neste momento
- Picat até o momento tem a tipagem **dinâmica**

Tipos de Dados

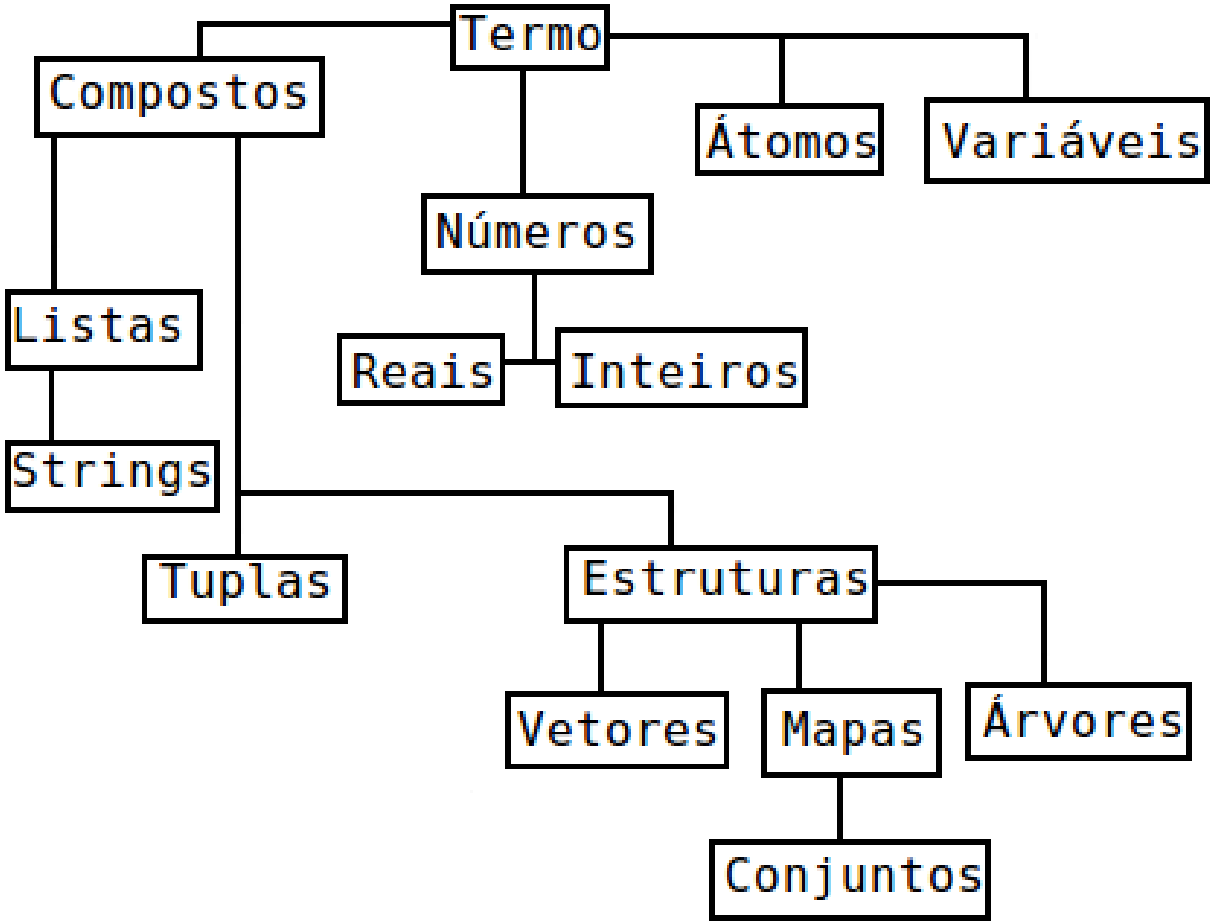


Figura 1: Hierarquia dos Tipos de Dados

Termos

- Em Picat, variáveis e valores são *genericamente* chamados de *termos*

Termos

- Em Picat, variáveis e valores são *genericamente* chamados de *termos*
- Os valores são subdivididos em duas categorias, números e valores compostos
- Os números, por suas vez, podem ser inteiros ou reais, e valores compostos podem ser listas e estruturas

Átomos

- Átomos são constantes simbólicas, podendo ser delimitados ou não, por aspas simples.
- Carácteres são representados por átomos de comprimento 1.
- Átomos não delimitados por aspas simples, **nunca** começam com uma letra maiúscula, nem número ou *underscore*.

Átomos

- Átomos são constantes simbólicas, podendo ser delimitados ou não, por aspas simples.
- Carácteres são representados por átomos de comprimento 1.
- Átomos não delimitados por aspas simples, **nunca** começam com uma letra maiúscula, nem número ou *underscore*.

Exemplos

```
x    x_1    '_'    '\\'    'a\'b\n'    '_ab'    '$%'
```

Números I

Números se dividem em:

- **Inteiro:** Inteiros podem ser representados por números binários, octais, decimais ou hexadecimais.

Exemplos

`12_345`	12345 em notação decimal, usando _ como separador
`0b100`	4 em notação binária
`0o73`	59 em notação octal
`0xf7`	247 em notação hexadecimal

O **underscore** é ignorado pelo compilador e o interpretador.

Números II

- **Real:** Números reais são compostos por um parte inteira, um ponto, seguido por uma fração decimal, ou um expoente.
- Se existe uma parte inteira em um número real então ela deve ser seguida por uma fração ou um expoente. Isso é necessário para distinguir um número real de um número inteiro.

Exemplos

```
12.345    0.123    12-e10    0.12E10
```

Compostos

- Termos compostos podem conter mais de um valor ao mesmo tempo.

Compostos

- Termos compostos podem conter mais de um valor ao mesmo tempo.
- Termos compostos são acessados pela notação de índice, começando a partir de 1 e indo até N, onde N é o tamanho deste termo.

Compostos

- Termos compostos podem conter mais de um valor ao mesmo tempo.
- Termos compostos são acessados pela notação de índice, começando a partir de 1 e indo até N, onde N é o tamanho deste termo.
- Se dividem em Listas e Estruturas.

Listas

Listas são agrupamentos de valores quaisquer sem ordem e sem tamanho pré-definido. Seu tamanho não é armazenado na memória, sendo necessário recalcular sempre que necessário seu uso. Listas são encapsuladas por colchetes.

Exemplos

[1,2,3,4,5] [a,b,32,1.5,aaac] ["string",14,22]

Há uma seção dedicada a esta poderosa estrutura de dados!

Strings – Lista de Carácteres

Strings são listas especiais que contém somente carácteres. *Strings* podem ser inicializadas como uma sequência de carácteres encapsulados por aspas duplas, ou como uma sequência de carácteres dentro colchetes separados por vírgulas.

Exemplos

"Hello" "World!" "\n" [o,l,a," ",m,u,n,d,o]

Tuplas

- Tuplas é um conjunto de termos não-ordenados, podendo ser acessados por notação de índice assim como listas.
- Tuplas são estáticas, ou seja, os termos contidos em uma tupla não podem ser alterados, assim como não podem ser adicionados ou removidos termos de tuplas.
- Tuplas são encapsuladas por parênteses e seus termos são separados por vírgulas.

Exemplos

(1,2,3,4,5) (a,b,32,1.5,aaac) ("string",14,22)

Em geral, usamos as tuplas dentro de listas.

Estruturas (*Functores*)

Estruturas são termos especiais que podem ser definidos pelo usuário. Estruturas tomam a seguinte forma:

$$\$s(t_1, \ldots, t_n)$$

Onde *'s'* é um átomo que denomina a estrutura, cada *'t_i'* é um de seus termos, e *'n'* é a aridade ou tamanho da estrutura.

Exemplo

$ponto(1,2) $pessoa(jose, "123.456.789.00", "1.234.567")

Estruturas (*Functores*)

Estruturas são termos especiais que podem ser definidos pelo usuário. Estruturas tomam a seguinte forma:

$$\$s(t_1, \ldots, t_n)$$

Onde *'s'* é um átomo que denomina a estrutura, cada *'t_i'* é um de seus termos, e *'n'* é a aridade ou tamanho da estrutura.

Exemplo

$ponto(1,2) $pessoa(jose, "123.456.789.00", "1.234.567")

Temos 4 outras estruturas que não usam o símbolo $, são elas:

Vetores

[fragile, allowframebreaks=0.9]

Vetores ou *arrays* são estruturas especiais do tipo:

$$\{t_1, \ldots, t_n\}$$

Vetores

[fragile, allowframebreaks=0.9]
Vetores ou *arrays* são estruturas especiais do tipo:

$$\{t_1, \ldots, t_n\}$$

- Vetor é um conjunto ordenado de tamanho n, delimitado por '{}'.
- Vetores tem comportamentos análogo às listas, tanto é que quase todas as funções de listas são sobrecarregadas para vetores.
- A diferença entre vetores e listas é que vetores tem um tamanho constante.
- Vetores são muito práticos quando se manipula matrizes na entrada

Exemplos

{1,2,3,4,5} {a,b,32,1.5,aaac} {"string",14,22}

Mapas, Conjuntos e *Heaps*

- **Mapas** são estruturas especiais que são conjuntos de relações do tipo `chave-valor`.
- **Conjuntos** são sub-tipos de mapas onde todos as chaves estão relacionadas com o átomo `not_a_value`.
- *Heaps* são árvores binárias completas representadas como vetores. Árvores podem ser do tipo *máximo*, onde o maior valor está na raiz, ou *mínimo*, onde o menor valor esta na raiz.

Variáveis I

- Picat é uma linguagem de Tipagem Dinâmica, ou seja, o tipo de uma variável é validado durante a execução do programa
- Isto é, quando uma variável é criada, seu tipo não é instanciado
- Variáveis são análogas as da matemática, são símbolos que *seguram* ou representam um valor
- Ao contrário de variáveis em linguagens imperativas, variáveis em Picat não são endereços simbólicos de locais na memória
- Uma variável é dita *livre* (*free*) se não contém nenhum valor, e dita *instanciada* (*bound*) se ela contém um valor
- Uma vez que uma variável é instanciada, ela permanece com este valor na execução atual
- Por isso, diz-se que variáveis em Picat são de *atribuição única*

Variáveis II

- O nome de variáveis devem sempre ser iniciado com letras **maiúsculas** ou com o caráctere ***underscore (_)***, porém;
  - Variáveis cujo nome é unicamente um caractere _ são chamadas de *variáveis anônimas*.
  - As *variáveis anônimas* podem receber qualquer valor não os guardam durante a execução do programa;
  - Num mesmo programa, podem existir diversas variáveis anônimas, instanciadas durante a execução do mesmo

Unficação e Atribuição

Há dois modos de definir valores às variáveis:

Unficação e Atribuição

Há dois modos de definir valores às variáveis:

- Unificação usa o operador $=$
- Atribuição usa o operador $:=$

Unificação

- A **Unificação** é uma operação que instancia uma variável a um termo, substituindo toda ocorrência dessa variável pelo valor

Unificação

- A **Unificação** é uma operação que instancia uma variável a um termo, substituindo toda ocorrência dessa variável pelo valor
- Caso ocorra uma instância que não falhe nenhuma situação a variável é unificada à este termo ou padrão.

Unificação

- A **Unificação** é uma operação que instancia uma variável a um termo, substituindo toda ocorrência dessa variável pelo valor
- Caso ocorra uma instância que não falhe nenhuma situação a variável é unificada à este termo ou padrão.
- Uma instanciação é indefinida até que se encontre um valor que possa ser unificada a uma variável.

Unificação

- A **Unificação** é uma operação que instancia uma variável a um termo, substituindo toda ocorrência dessa variável pelo valor
- Caso ocorra uma instância que não falhe nenhuma situação a variável é unificada à este termo ou padrão.
- Uma instanciação é indefinida até que se encontre um valor que possa ser unificada a uma variável.
- Termos são ditos unificáveis se são idênticos ou podem ser tornados idênticos instanciado variáveis nos termos.

Exemplo

```
Picat> X = 1
X = 1
Picat> $f(a,b) = $f(a,b)
yes
Picat> [H|T] = [a,b,c]
H = a
T = [b,c]
Picat> $f(X,b) = $f(a,Y)
X = a
Y = b
Picat> bind_vars({X,Y,Z},a)
Picat> X = $f(X)
```

Cuidar neste último caso, há um laço infinito nesta chamada!

Atribuição

- A **atribuição** simula a atribuição em linguagens imperativas

Atribuição

- A **atribuição** simula a atribuição em linguagens imperativas
- Permite que variáveis assumam novos valores durante a execução do programa

Atribuição

- A **atribuição** simula a atribuição em linguagens imperativas
- Permite que variáveis assumam novos valores durante a execução do programa
- O escopo da atribuição da variável é local e volátil

Atribuição

- A **atribuição** simula a atribuição em linguagens imperativas
- Permite que variáveis assumam novos valores durante a execução do programa
- O escopo da atribuição da variável é local e volátil
- Na unificação, uma nova variável temporária é criada afim de substituir um valor atribuído ou outra variável.

Exemplo

```
test => X = 0, X := X + 1,  X := X + 2, write(X).
```

- Neste exemplo X é unificado a 0.
- Em seguida, há uma atribuição X a $X + 1$, porém X já foi unificado a um termo.
- Então, outras operações devem ser feitas para que esta atribuição seja possível.
- Nesse caso, o compilador cria uma variável temporária, $X1$ por exemplo, e unifica com $X + 1$. Cada vez que X for instanciado, o compilador/programa atualiza em $X1$.
- O mesmo ocorre na atribuição $X1 := X1 + 2$, neste caso uma outra variável temporária é criada, por exemplo $X2$, e o processo se repetido.

Exemplo

```
test => X = 0, X := X + 1,  X := X + 2, write(X).
```

- Neste exemplo X é unificado a 0.
- Em seguida, há uma atribuição X a $X + 1$, porém X já foi unificado a um termo.
- Então, outras operações devem ser feitas para que esta atribuição seja possível.
- Nesse caso, o compilador cria uma variável temporária, $X1$ por exemplo, e unifica com $X + 1$. Cada vez que X for instanciado, o compilador/programa atualiza em $X1$.
- O mesmo ocorre na atribuição $X1 := X1 + 2$, neste caso uma outra variável temporária é criada, por exemplo $X2$, e o processo se repetido.

Portanto, estas atribuições sucessivas são compiladas como:

Exemplos de Variáveis Válidas

X1	_	_ab
X	A	Variavel
_invalido	_correto	_aa

Relembrando, um nome de variável é válido se começa com letra **maiúscula** ou _

Exemplos de Variáveis Inválidas

1_Var	variável	valida
23	"correto	‘termo
!numero	$valor	#comum

Tabela 1: Operadores Aritméticos em Ordem de Precedência

X ** *Y*	Potenciação
X * *Y*	Multiplicação
X / *Y*	Divisão, resulta em um real
X // *Y*	Divisão de Inteiros, resulta em um inteiro
X `mod` *Y*	Resto da Divisão
X + *Y*	Adição
X - *Y*	Subtração
Inicio .. *Passo* .. *Fim*	Uma série (lista) de números com um passo
Inicio .. *Fim*	Uma série (lista) de números com passo 1

REDIMENSIONAR

Tabela 2: Tabela de Operadores Completa em Ordem de Precedência

Operadores Aritméticos	Ver Tabela 1
++	Concatenação de Listas/Vetores
= :=	Unificação e Atribuição
== =:=	Equivalência e Equivalência Numérica
!= !==	Não Unificável e Diferença
< =< <=	Menor que
> >=	Maior que
in	Contido em
not	Negação Lógica
, &&	Conjunção Lógica
; ||	Disjunção Lógica

Operadores Especiais I

Operadores de Termos Não Compostos

Operadores Especiais II

1. **Equivalência**(==): Compara se dois termos são iguais. No caso de termos compostos, eles são ditos equivalentes se todos os termos contidos em si são equivalentes. O compilador considera termos de tipos diferentes como totalmente diferentes, portanto a comparação $1.0 == 1$ seria avaliada como falsa, mesmo que os valores sejam iguais. Nesses casos, usa-se a *Equivalência Numérica*.
2. **Equivalência Numérica**(=:=): Compara se dois números são o mesmo valor. Não deve ser usada com termos que não são números.
3. **Diferença**(!==): Compara se dois termos são diferentes. Mesmo que a negação da equivalência.
4. **Não Unificável**(!=): Verifica se dois termos não são unificáveis. Termos são ditos unificáveis se são idênticos ou podem ser tornados idênticos instanciando variáveis destes

Programação por Restrições (PR) – I

- A Programação por Restrições (PR) é conhecida por *Constraint Programming* ou simplesmente **CP**

Programação por Restrições (PR) – I

- A Programação por Restrições (PR) é conhecida por *Constraint Programming* ou simplesmente **CP**
- Uma poderosa teoria (e técnica) que contorna a complexidade de certos problemas exponenciais

Programação por Restrições (PR) – I

- A Programação por Restrições (PR) é conhecida por *Constraint Programming* ou simplesmente **CP**
- Uma poderosa teoria (e técnica) que contorna a complexidade de certos problemas exponenciais
- A PR encontrava-se inicialmente dentro da IA e PO, mas como várias outras, tornaram-se fortes e autônomas. Atualmente uma área de pesquisa bem forte em alguns países.

Programação por Restrições (PR) – II

- Aproximadamente o algoritmo da PR é dado:

Programação por Restrições (PR) – II

- Aproximadamente o algoritmo da PR é dado:
  1. Avaliar algebricamente os domínios das variáveis com suas restrições
  2. Intercala iterativamente a propagação de restrições com um algoritmo de busca
  3. A cada variável instanciada, o processo é repetido sobre as demais variáveis, reduzindo progressivamente o espaço de busca
  4. Volte ao passo inicial até que os domínios permaneçam estáticos e que as variáveis apresentem instâncias consistentes

Programação por Restrições (PR) – II

- Aproximadamente o algoritmo da PR é dado:
  1. Avaliar algebricamente os domínios das variáveis com suas restrições
  2. Intercala iterativamente a propagação de restrições com um algoritmo de busca
  3. A cada variável instanciada, o processo é repetido sobre as demais variáveis, reduzindo progressivamente o espaço de busca
  4. Volte ao passo inicial até que os domínios permaneçam estáticos e que as variáveis apresentem instâncias consistentes
- Este núcleo é uma busca por constantes otimizações

Programação por Restrições (PR) – II

- Aproximadamente o algoritmo da PR é dado:
  1. Avaliar algebricamente os domínios das variáveis com suas restrições
  2. Intercala iterativamente a propagação de restrições com um algoritmo de busca
  3. A cada variável instanciada, o processo é repetido sobre as demais variáveis, reduzindo progressivamente o espaço de busca
  4. Volte ao passo inicial até que os domínios permaneçam estáticos e que as variáveis apresentem instâncias consistentes
- Este núcleo é uma busca por constantes otimizações
- Uma das virtudes da PR: a legibilidade e clareza de suas soluções

Programação por Restrições (PR) – III

- Problemas combinatoriais com domínio nos inteiros são bons candidatos a serem resolvidos por PR

Programação por Restrições (PR) – III

- Problemas combinatoriais com domínio nos inteiros são bons candidatos a serem resolvidos por PR
- Quando temos problemas que precisamos conhecer **todas** as respostas, não apenas a melhor resposta

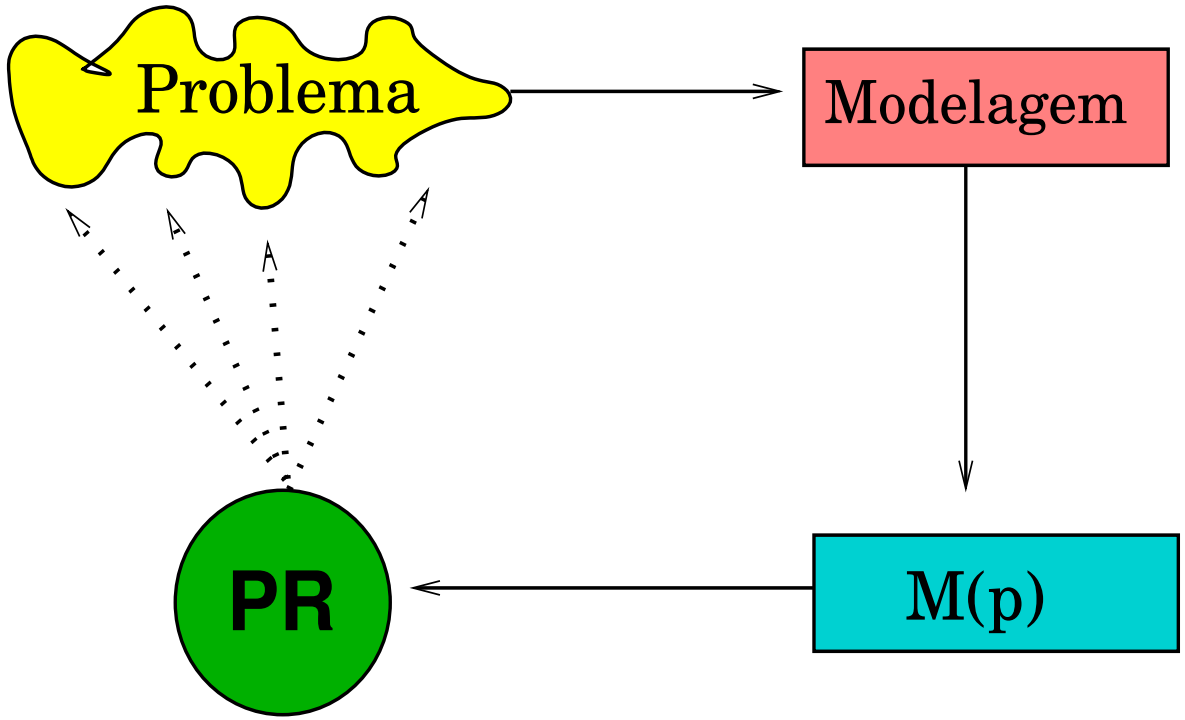
Programação por Restrições (PR) – III

- Problemas combinatoriais com domínio nos inteiros são bons candidatos a serem resolvidos por PR
- Quando temos problemas que precisamos conhecer **todas** as respostas, não apenas a melhor resposta
- Quando necessitamos de respostas *precisas* e não apenas as aproximadas. Há um custo computacional a ser pago aqui!

Programação por Restrições (PR) – III

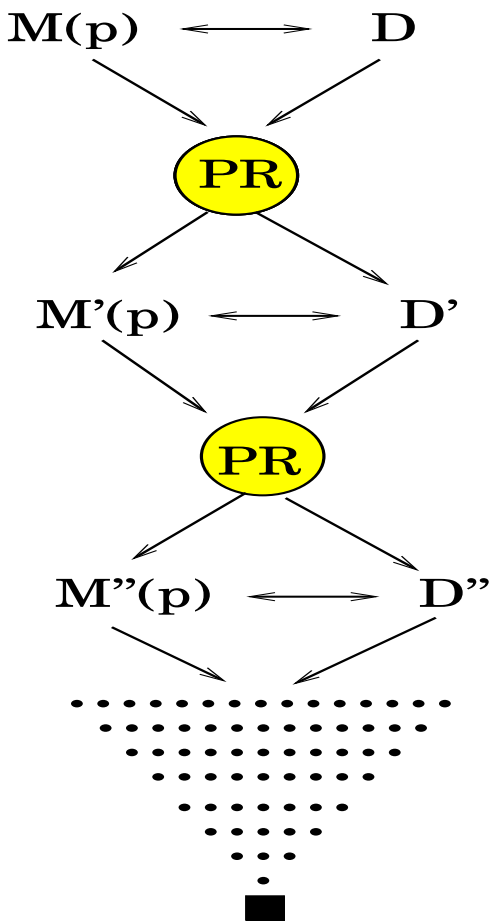
- Problemas combinatoriais com domínio nos inteiros são bons candidatos a serem resolvidos por PR
- Quando temos problemas que precisamos conhecer **todas** as respostas, não apenas a melhor resposta
- Quando necessitamos de respostas *precisas* e não apenas as aproximadas. Há um custo computacional a ser pago aqui!
-

Metodologia da Construção de Modelos



Um modelo

Fluxo de Cálculo da PR



Onde o objetivo da PR é:

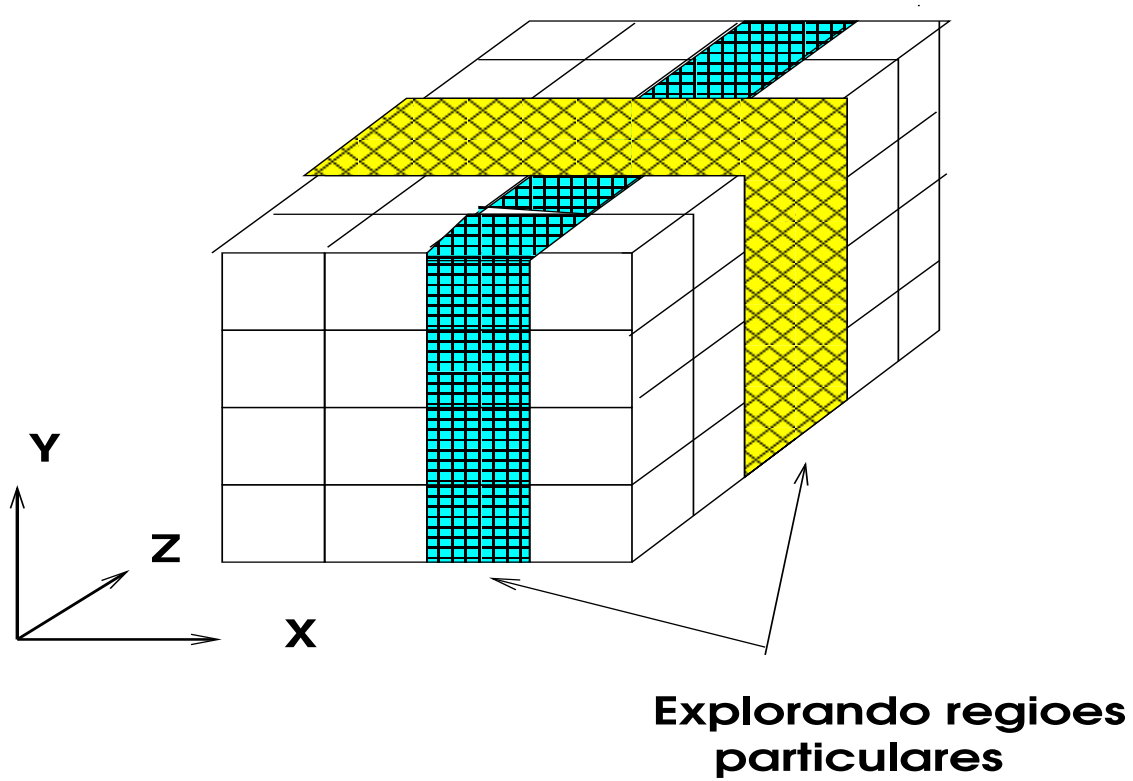


Figura 2: Realizar buscas com regiões reduzidas – promissoras (regiões factíveis de soluções)

Redução Iterativa em Sub-problemas

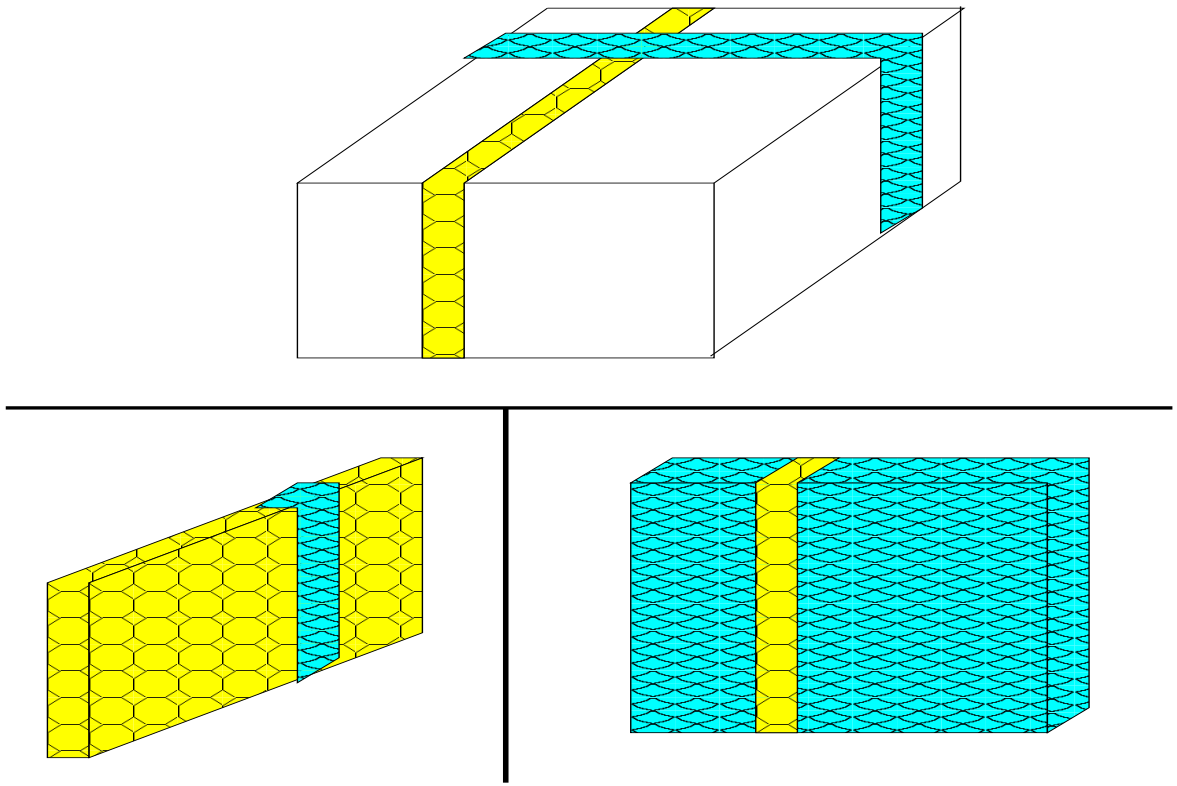


Figura 3: Redução de um CP em outros sub-problemas CPs equivalentes

Exemplo

- Dado um número par qualquer, encontre dois de números primos, N_1 e N_2, diferentes entre si, que somados deêm este número par.

Exemplo

- Dado um número par qualquer, encontre dois de números primos, N_1 e N_2, diferentes entre si, que somados deêm este número par.
- Exemplo:
  Seja o PAR = 18
  Uma soluç ao:
  $N_1 = 7$ e $N_2 = 11$
  pois
  $N_1 + N_2 = 18$

Modelagem do Problema

- N_1 e N_2 assumem valores no domínio dos números primos. Logo, é importante ter os números primos prontos!

Modelagem do Problema

- N_1 e N_2 assumem valores no domínio dos números primos. Logo, é importante ter os números primos prontos!
- A soma destes números é o par fornecido como entrada, N_{PAR}: $N_1 + N_2 = N_{PAR}$

Modelagem do Problema

- N_1 e N_2 assumem valores no domínio dos números primos. Logo, é importante ter os números primos prontos!
- A soma destes números é o par fornecido como entrada, N_{PAR}: $N_1 + N_2 = N_{PAR}$
- N_1 e N_2 são diferentes entre si $N_1 \neq N_2$

Modelagem do Problema

- N_1 e N_2 assumem valores no domínio dos números primos.
  Logo, é importante ter os números primos prontos!
- A soma destes números é o par fornecido como entrada, N_{PAR}:
  $N_1 + N_2 = N_{PAR}$
- N_1 e N_2 são diferentes entre si
  $N_1 \neq N_2$
- Como são inteiros: $N_1 < N_{PAR}$ e $N_2 < N_{PAR}$
  Sim, é óbvio, mas isto faz uma redução significativa de domínio!

Código Completo

- Acompanhar as explicações do código de: `https://github.com/claudiosa/CCS/blob/master/picat/soma_N1_N2_primos_CP.pi`
- Confira a execução e testes

Código em Partes

```
modelo =>
    PAR = 382,
    Variaveis = [N1,N2],
    % Gerando um domino soh de primos
    % L_dom = [I : I in 1..1000, eh_primo(I) == true],   %OU
    L_dom = [I : I in 1..1000, prime(I)],
    Variaveis :: L_dom,
```

Uma ótima estratégia: sair com um domínio de números candidatos!

Código em Partes

```
    % RESTRICOES
    N1 #!= N2,
    N1 #< PAR,
    N2 #< PAR,
    N1 + N2 #= PAR,

% A BUSCA
solve([ff], Variaveis),
  % UMA SAIDA
printf("\n  N1: %d\t N2: %d", N1,N2),
printf("\n.....................................")
.
```

Código em Partes

```
import cp.

% main => modelo .
% main ?=> modelo, fail.
% main =>  true.

main =>
    L = findall(_, $modelo),
    writef("\n Total de solucoes:  %d \n", length(L)) .
```

Saída – I

```
Picat> cl('soma_N1_N2_primos_CP').
Compiling:: soma_N1_N2_primos_CP.pi
** Warning  : redefine_preimported_symbol(math): prime / 1
soma_N1_N2_primos_CP.pi compiled in 7 milliseconds
loading...

yes

Picat> main.

  N1: 3  N2: 379
.....................................
  N1: 23  N2: 359
.....................................
  N1: 29  N2: 353
.....................................
```

Saída – II

```
 ....................................
  N1: 353  N2: 29
....................................
  N1: 359  N2: 23
....................................
  N1: 379  N2: 3
....................................
 Total de solucoes:  18

yes

Picat>
```

Reflexões

- Há outros métodos para se resolver estes problemas. Exemplo: Programação Linear, Buscas Heurísticas, etc

Reflexões

- Há outros métodos para se resolver estes problemas.
  Exemplo: Programação Linear, Buscas Heurísticas, etc
- A área é extensa, contudo, Picat adere há todos requisitos da PR

Reflexões

- Há outros métodos para se resolver estes problemas. Exemplo: Programação Linear, Buscas Heurísticas, etc
- A área é extensa, contudo, Picat adere há todos requisitos da PR
- Resumo da PR: segue por uma notação/manipulação algébrica restrita, simplificar e bissecionar as restrições, instanciar variáveis, verificar inconsistências, avançar sobre as demais variáveis, até que todas estejam instanciadas.