INTERPRETADOR HOPE - REFERÊNCIA[1]

Ross Paterson, City, University of London, <ross@soi.city.ac.uk>
Tradução e adaptação[2] por Augusto Manzano, IFSP, <augusto.garcia@ifsp.edu.br>

Este manual não se trata de um tutorial de programação funcional, nem tão pouco sobre como fazer uso da linguagem Hope. Caso não conheça os dois, pode começar com algo como o tutorial de Roger Bailey. Também traduzido e disponibilizado em ***www.hope.manzano.pro.br[3]****.*

SUMÁRIO

[1] Tradução realizada a partir do artigo depositado em: *http://www.stolyarov.info/static/hope/ref_man/ref_man.html*.

[2] Esta tradução foi adaptada para melhor legibilidade a todos os públicos. Alguns pequenos ajustes são definidos para melhor compatibilizar o texto com o idioma português. Esta tradução é mantida na segunda pessoa do singular e nela foram acrescidos detalhes complementares que não estavam no documento original.

[3] Observação do tradutor.

ESTRUTURA LEXICA

A entrada é dividida em uma sequência de símbolos dos seguintes tipos:

- um caractere de pontuação: aspa simples {'}, parênteses {()}, vírgula {,}, ponto e vírgula {;}, colchetes {[]} e dois pontos {:};

- um identificador:

    1. uma letra ou sublinhado seguido por zero ou mais letras, sublinhados ou dígitos (seguido por zero ou mais aspas simples), ou

    2. uma sequência de caracteres gráficos que não são espaços em branco, letras, sublinhados, dígitos, pontuação nem um dos símbolos crase {`}, exclamação {!}, aspa inglesa {"} e mais de uma aspa simples {' ' '};

- um literal numérico: uma sequência de dígitos (em alguns sistemas opcionalmente com um ponto decimal embutido e um expoente opcionalmente assinado);

- um literal de caractere: um único caractere (mas não uma nova linha) ou uma sequência de escape de caractere (como na linguagem C), entre aspas simples ou;

- um literal de cadeia (*string*): uma sequência de zero ou mais caracteres (mas não novas linhas de aspas duplas) ou sequências de escape de caractere, entre aspas inglesas.

Os símbolos podem ser separados por espaço em branco e comentários. Um comentário é introduzido com o símbolo "!" (exclamação) e continua até o fim da linha.

Os seguintes identificadores são reservados pela linguagem Hope:

```
++ --- : <= == => |
abstype data dec display else error[4] edit exit help[5] id
if in infix infixr lambda let letrec mu private read
save then type typevar uses where whererec write
```

Os seguintes identificadores são também reservados, para compatibilidade com outras implementações da linguagem:

```
end module nonop pubconst pubfun pubtype
```

Além disso, os seguintes sinônimos estão disponíveis: "\" para "lambda", "use" para "uses" e "infixrl" para "infixr".

[4] Os comandos: error, help, id, mu e read foram acrescentados pelo tradutor.
[5] Anexado pelo tradutor - somente existe nas versões Hope para o sistema operacional Windows.

IDENTIFICADORES

Os identificadores podem referir-se a:

- módulos,
- tipos,
- construtores de tipo,
- constantes de dados,
- construtores de dados, ou
- valores.

O mesmo identificador pode referenciar mais de um desses tipos, mas não pode referenciar a mais de uma das três últimas classes.

Identificadores podem ser definidos como operadores "infix <operadores>", por "infix <definições>" e "infixr <definições>". Isso afeta apenas a maneira como são analisados e impressos: se um identificador "@" for declarado como *infix* (infixo) as construções da forma "@(a , b)" passam a ser escritas como “a @ b”. Para fazer referência apenas ao símbolo "@" use " (@) ". Um caso especial: em qualquer declaração subsequente de "@" como um construtor de dados ao estilo "@(tipo_ 1 # tipo_ 2) " deve ser escrito "tipo_ 1 @ tipo_ 2".

Vários identificadores são predefinidos no módulo Standard da linguagem.

ESTRUTURAS COMPOSTAS

Na descrição a seguir os termos "constataste", "largura", "texto" (por exemplo, "dec" ou "++") denotam a definição de textos literais e estarão grafados em negrito, enquanto texto grafado em itálico (por exemplo, *identificador de tipo* ou *padrão*) denotam classes sintáticas. Elementos sem terminação são declarados em seções com o mesmo nome. Os colchetes "[" e "]" são usados em torno de texto que seja opcional. Formulários alternativos são usados se os identificadores forem declarados como operadores.

Módulos

O nome do módulo de biblioteca de funcionalidades para a linguagem Hope é um nome de arquivo em formato texto com a extensão ".hop”, o qual consiste em um conjunto de definições. As definições podem ocorrer em qualquer ordem, mas os identificadores devem ser declarados apropriadamente antes de seu uso.

Há um módulo Hope especial chamado "Standard.hop" que é importado automaticamente por todos os outros módulos e sessões da linguagem. Este arquivo não deve ser alterado ou modificado sob risco de prejudicar o funcionamento da linguagem[6].

Uma sessão interativa consiste em definições realizadas juntamente com outros comandos.

[6] Frase acrescentada pelo tradutor.

Definições

uses / **use** *identificador-de-módulo, ..., identificador-de-módulo*;

Todas as definições visíveis nos módulos nomeados devem estar visíveis no módulo ou sessão atual. Os módulos são procurados primeiro no diretório atual e, em seguida, em um diretório de biblioteca. Módulos usados podem usar outros módulos, e assim por diante, desde que não haja referência cíclica entre eles. As definições visíveis no módulo (ou sessão) em uso não são visíveis em um módulo em uso, a menos que as definições residam em um terceiro módulo usado por ambos.

private;

todas as definições subsequentes neste módulo serão ocultadas de outros módulos que o utilizam. Em particular, as definições visíveis em um módulo usado após esta diretiva não serão visíveis para nenhum módulo usando esta, ao passo que estariam se o módulo fosse usado antes da diretiva.

infix *identificador, ..., identificador* : *literal-numérica*;

Declara a definição de operadores infixos associados à esquerda com a precedência estabelecida entre os níveis de 1 (mais fraco) até 9 (mais forte).

infixr *identificador, ..., identificador* : *literal-numérica*;

Declara a definição de operadores infixos associados à direita com a precedência estabelecida entre os níveis de 1 (mais fraco) até 9 (mais forte).

abstype *identificador* [(*identificador_ 1, ..., identificador_ n*)];

Uma definição de tipo abstrato declarando o *identificador* como um tipo ou identificador de construtor de tipos que pode ser usado em definições subsequentes. O *identificador* pode ser estabelecido posteriormente por uma definição de "tipo" ou "dados". Se houver apenas um argumento os parênteses são opcionais.

data *identificador* [(*identificador_ 1, ..., identificador_ n*)] ==
    identificador_ 1' [*tipo_ 1*] ++ ... ++ *identificador_ k'* [*tipo_ k*];

Define o *identificador* como sendo um tipo ou construtor de tipos com o *identificador_ k'* como suas constantes de dados e construtores. Se houver apenas um argumento simples os parênteses são opcionais.

Qualquer identificador de construtor pode ter sido declarado anteriormente em uma declaração "dec", caso em que o novo tipo de identificador deve ser uma instância daquele fornecido na declaração anterior.

type *identificador* [(*identificador_ 1, ..., identificador_ n*)] == *tipo*;

Define o *identificador* como uma abreviatura para o *tipo* que pode se referir aos identificadores de tipo de argumento. Se houver apenas um argumento simples os parênteses são opcionais.

A definição pode ser recursiva. Qualquer uso de *identificador* no *tipo* deve ter os mesmos parâmetros do lado esquerdo. Consulte *os tipos regulares* para obter mais detalhes.

typevar *identificador, ..., identificador*;

Declara novas variáveis de tipo para uso em declarações "dec".

Se o *identificador* for declarado então o *identificador'* , *identificador''* e assim por diante também podem ser usados como variáveis de tipo (aspas simples no próprio *identificador* são ignoradas).

dec *identificador, ..., identificador* : *tipo*;

Declara os identificadores como identificadores de valor do tipo dado em que as variáveis de tipo declaradas pelas definições de *typevar* representam qualquer tipo.

Se apenas um identificador estiver sendo declarado a palavra-chave "dec" é opcional.

--- *padrão-de-identificador-de-valor_ 1 ... padrão_ n* <= *expressão*;

Definir o valor de um identificador. Se "n" for zero, apenas uma dessas definições é permitida para cada identificador. Caso contrário, o identificador pode ser definido por uma ou mais definições cada uma com o mesmo número de argumentos. Consulte o tópico *SEMÂNTICA DE CORRESPONDÊNCIA DE PADRÕES*.

A sequência de símbolos "---" é opcional.

Tipos

tipo

O tipo referido.

tipo-construtor

Um tipo construído ou uma abreviatura de tipo. Se houver apenas um argumento para o tipo os parênteses são opcionais.

(*_tipo*)

Igual ao *tipo*.

Padrões

identificador

Define uma variável que pode ser associada a qualquer valor. Nenhuma variável poderá ocorrer mais de uma vez no mesmo padrão. Cada ocorrência livre do *identificador* na expressão correspondente ao padrão é limitada por esse padrão, ou seja, o identificador é um *identificador-de-valor* na expressão referindo-se ao valor correspondido. Este identificador não se aplica a um *identificador-de-constante-de-dados*.

constante-de-dados

Corresponde à constante de dados nomeada.

padrão-de-construtor-de-dados

Corresponde a um valor formado pela aplicação do construtor de dados a um padrão de correspondência de valor.

literal-numérica

"1", "2", "..." são abreviações para "succ(0) ", "succ(succ(0)) ", etc.

padrão + literal-numérica

O formato do "padrão + k" é uma abreviatura de "succ" aplicado a "k" vezes ao "padrão".

'c'

Corresponde à constante do caractere.

"cadeia" (string)

Abreviatura para uma lista de caracteres (conjunto de caracteres).

[padrão_ 1, ..., padrão_ n]

Equivalente a "padrão_ 1 :: ... :: padrão_ n :: nil".

[]

Equivalente a "nil".

padrão_ 1, padrão_ 2

Corresponde a um par de valores associados ao "padrão_ 1" e ao "padrão_ 2" respectivamente.

(_padrão)

O mesmo que "padrão".

Expressões

valor

Um valor vinculado ao identificador.

constante-de-dados

Uma constante de dados.

construtor-de-dados

Um construtor de dados.

literal-numérica

"1", "2", "..." são abreviações para "succ(0) ", "succ(succ(0)) ", etc.
'c'

Constante de caractere.

"cadeia" (string)

Abreviatura para uma lista de caracteres (conjunto de caracteres).

[padrão_ 1, ..., padrão_ n]

Equivalente a "padrão_ 1 :: ... :: padrão_ n :: nil".

[]

Equivalente a "nil".

expressão_ 1, expressão_ 2

Corresponde a um par de valores associados a "expressão_ 1" e a "expressão_ 2" respectivamente.

(_expressão)

O mesmo que "expressão".

expressão_ 1 expressão_ 2

O resultado da aplicação da função ou valor do construtor da expressão_ 1" ao valor da "expressão_ 2"

lambda padrão_ 1 => expressão_ 1 | ... | padrão_ k => expressão_ k

Uma função anônima. Consulte o tópico *SEMÂNTICA DE CORRESPONDÊNCIA DE PADRÕES*.

if expressão_ 1 then expressão_ 2 else expressão_ 3

Uma expressão condicional, igual à "expressão_ 2" se a "expressão_ 1" for verdadeira ou a "expressão_ 3" se for falsa.

let padrão == expressão_ 1 in expressão_ 2

O mesmo que a "expressão_ 2" com variáveis na "expressão_ 2" substituídas pelos valores atribuídos a elas no padrão de correspondência da "expressão_ 1". Equivalente a

```
  (lambda
padrão
  =>
expressão_ 2
  ) (fix(lambda
padrão
     =>
expressão_ 1
     ))
```

letrec padrão == expressão_ 1 in expressão 2

Como "let", exceto que o padrão deve ser irrefutável e suas variáveis podem aparecer na "expressão_ 1". É uma versão mais eficiente de

```
  (lambda
padrão
  =>
expressão_ 2
  ) (fix(lambda
padrão
     =>
expressão_ 1
     ))
```

para a função de correção "fix" definida como

```
dec fix : (alpha -> alpha) -> alpha;
--- fix f <= f (fix f);
```

expressão_ 1 where padrão == expressão_ 2

O mesmo que "let padrão == expressão_ 1 in expressão_ 2"'.

expressão_ 1 whererec padrão == expressão_ 2

O mesmo que "letrec padrão == expressão_ 2 in expressão_ 1".

Ambiguidades em padrões e expressões são resolvidas pelas seguintes precedências de ligação, da mais fraca para a mais forte:

- *vírgula* (associativa à direita)
- *lambda*
- *let* e *letrec*
- *where* e *whererec*
- *if*
- operador *infix* de precedência 1 a 9
- *aplicação de função* (associativa à esquerda)

Para qualquer operador infixo "@" e expressão "e" as seguintes abreviações, chamadas seções de operador, são permitidas:

(e @)

Como abreviação de "lambda x => e @ x".

(@ e)

Como abreviação de "lambda x => x @ e".

Comandos interativos

Em uma sessão Hope, quaisquer definições podem ser inseridas (embora privado seja ignorado), bem como os seguintes comandos:

expressão

Exibe o valor e o tipo de *expressão* mais geral. A avaliação preguiçosa é usada, mas nada é exibido até que o valor seja totalmente avaliado.

A *expressão* pode envolver a entrada de variável especial que denota a lista de caracteres informadas no terminal.

write expressão [to "arquivo"];

Produz o valor da *expressão* (que deve ser uma lista de valores) diretamente: se o valor for uma lista de caracteres, os caracteres são impressos; caso contrário, cada valor é impresso em uma linha por si só. Cada elemento da lista é impresso assim que seu valor é calculado (por avaliação preguiçosa). Se a cláusula "to" estiver presente a saída será gravada em arquivo; caso contrário, será impresso na tela. É muito seguro gravar em um arquivo que foram lidos pela expressão, sendo que nenhuma interferência ocorrerá.

A *expressão* pode envolver a entrada de variável especial que denota a lista de caracteres informadas no terminal.

display;

Exibe na tela as definições da sessão atual.

save identificador-de-modulo;

Grava as definições da sessão atual em um novo módulo com o nome fornecido. As definições são substituídas na sessão atual por uma referência ao novo módulo.

edit [identificador-de-modulo];

(Somente disponível para sistemas operacionais padrão POSIX - Unix). Tem por finalidade invocar o editor de texto padrão no módulo nomeado, se o nome for fornecido, ou de outra forma em um arquivo contendo as definições atuais para armazená-las ao uso posterior.

exit;

Encerra a execução do interpretador.

SEMÂNTICA DE CORRESPONDÊNCIA DE PADRÕES

A seguir encontra-se uma especificação parcial não sendo garantido nada além disso Informalmente quando os padrões se sobrepõem. Padrões mais específicos são preferidos a outros mais específicos. Além disso, a escolha não é especificada.

A correspondência de um valor com um padrão envolve avaliar o valor o suficiente para comparar suas constantes de dados e construtores com aqueles no padrão, em alguma ordem consistente de cima para baixo (mas não especificada). Esta correspondência pode resultar em:

```
bem sucedida (success):
```

o valor pode ser visto como uma instância do padrão,

```
fracassada (failure):
```

o valor não pode ser uma instância do padrão porque tem uma constante de dados diferente ou construtor em alguma posição e

```
não encerrada (non-termination):
```

o valor não pode ser avaliado suficientemente para decidir entre os acima que estejam na ordem particular de verificação usada.

Se a correspondência for bem-sucedida serão fornecidos os valores para as variáveis do padrão.

Um padrão (ou subpadrão) consiste apenas em variáveis e pares sempre correspondendo a um valor do tipo apropriado, mesmo que o cálculo do valor não termine. Esses padrões são irrefutáveis.

Agora, suponha que uma função tenha sido definida por uma série de definições

```
  --
f
p
  _
i1
...
p
  _
in
  <=
e
  _
i
;

for i = 1, ..., k, or is the expression

  lambda
p
  _
11
  ...
p
  _
1n
  =>
e
  _
1
  |
...
  |
```

```
p
  _
k1
...
p
  _
kn
  =>
e
  _
k
```

(Neste momento as funções anônimas devem ser unárias.) Uma aplicação desta função a "n" valores de argumento terá zero ou mais valores possíveis, como segue:

- Se alguns padrões "p_ ij" corresponder com sucesso aos argumentos definidos para algum "i", e todos os padrões mais específicos (se houver) falharem um valor possível é o valor de "e_ i" com as variáveis de "p_ ij" assumindo os valores correspondentes.

- Se a correspondência de algum padrão não terminar, sob alguma ordem de verificação, então a ocorrência "não-rescisão" é um valor possível.

Se não houver valores possíveis o valor da ação é um erro em tempo de execução. Caso contrário, um dos valores possíveis é escolhido de forma determinística (mas não especificado).

APÊNDICE: O MÓDULO PADRÃO

O ambiente padrão de Hope usado implicitamente por cada sessão e módulo.

- Construtores de tipo padrão
- Funções definidas internamente

Construtores do tipo padrão

Variáveis de tipos padrão.

```
typevar alfa, beta, gamma;
```

Função e tipos de produto.

```
infixr -> : 2;
abstype neg -> pos;

infixr # : 4;
abstype pos # pos;

--- (a # b) (x, y) <= (a x, b y);

infixr X : 4;
type alpha X beta == alpha # beta;
```

Composição normal da direita para a esquerda de "f" e "g".

```
infix o : 2;
dec o : (beta -> gamma) # (alpha -> beta) -> alpha -> gamma;
--- (f o g) x <= f(g x);

--- (a -> b) f <= b o f o a;
```

A função identidade.

```
dec id : alpha -> alpha;
--- id x <= x;
```

Operações padrão com ações lógicas (booleanas).

```
data bool == false ++ true;
type truval == bool;

infix or : 1;
infix and : 2;

dec not : bool -> bool;
--- not true <= false;
--- not false <= true;

dec and : bool # bool -> bool;
--- false and p <= false;
--- true and p <= p;

dec or : bool # bool -> bool;
--- true or p <= true;
--- false or p <= p;

dec if_then_else : bool -> alpha -> alpha -> alpha;
--- if true then x else y <= x;
--- if false then x else y <= y;
```

Listas.

```
infixr :: : 5;
data list alpha == nil ++ alpha :: list alpha;
```

Concatenação de lista.

```
infixr <> : 5;
dec <> : list alpha # list alpha -> list alpha;
--- [] <> ys <= ys;
--- (x::xs) <> ys <= x::(xs <> ys);
```

O tipo "num" se comporta como se fosse definido por:

```
!        data num == 0 ++ succ num;
```

mas, na verdade é implementado com um pouco mais de eficiência. O tipo também contém números negativos (e reais em algumas implementações).

```
data num == succ num;
```

Da maneira similar, o tipo "char" se comporta como se fosse definido como uma enumeração (na ordem normal) de todas as constantes de caracteres ASCII.

```
abstype char;
--- char x <= x;
```

Funções definidas internamente

Funções de comparação: avaliam seus argumentos apenas na medida necessária para determinar sua ordem. Ao comparar constantes e construtores distintos o menor é aquele que veio antes na definição de dados em que ambos foram definidos. Os pares são comparados lexicograficamente enquanto a comparação de funções aciona um erro em tempo de execução. Nos tipos "num", "char" e "list (char) " essas regras fornecem as ordens normais.

```
infix =, /= : 3;
infix <, =<, >, >= : 4;
dec =, /=, <, =<, >, >= : alpha # alpha -> bool;
```

Funções de comparação de nível inferior.

```
data relation == LESS ++ EQUAL ++ GREATER;

dec compare : alpha # alpha -> relation;

--- x =  y <= (\ EQUAL => true | _ => false) (compare(x, y));
--- x /= y <= (\ EQUAL => false | _ => true) (compare(x, y));
--- x <  y <= (\ LESS => true | _ => false) (compare(x, y));
--- x >= y <= (\ LESS => false | _ => true) (compare(x, y));
--- x >  y <= (\ GREATER => true | _ => false) (compare(x, y));
--- x =< y <= (\ GREATER => false | _ => true) (compare(x, y));
```

Conversões.

```
dec ord : char -> num;
dec chr : num -> char;

dec num2str : num -> list char;
dec str2num : list char -> num;
```

O conteúdo de um arquivo nomeado (criado preguiçosamente).

```
dec read : list char -> list char;
```

Aborta a execução com uma mensagem de erro.

```
dec error : list char -> alpha;
```

Funções aritméticas usuais.

```
infix +, - : 5;
infix *, /, div, mod : 6;
dec +, -, *, /, div, mod : num # num -> num;
```

Biblioteca matemática.

```
dec cos, sin, tan, acos, asin, atan : num -> num;
dec atan2, hypot : num # num -> num;
dec cosh, sinh, tanh, acosh, asinh, atanh : num -> num;
dec abs, ceil, floor : num -> num;
dec exp, log, log10, sqrt : num -> num;
dec pow : num # num -> num;
dec erf, erfc : num -> num;
```

Quaisquer argumentos extras na linha de comando do interpretador.

```
dec argv : list(list char);
```

Parte do tratamento especial do construtor de "succ".

```
dec succ : num -> num;
--- succ n <= n + 1;
```

DESVIOS DE OUTRAS VERSÕES DA LINGUAGEM

Os recursos a seguir são especiais para esta versão Hope: padrões irrefutáveis, seções de operador, funções de mais de um argumento, definições de tipo recursivo, entrada, leitura, gravação, privado, abstype, letrec e whererec. Outras versões da linguagem podem não suportar avaliação completa e lenta e podem ter vários recursos não fornecidos aqui.

Esta versão também suporta construtores de tipo e dados "curried".

O restante deste apêndice descreve alguns recursos experimentais desta versão da linguagem.

Tipos regulares

O sistema de tipos é generalizado para permitir tipos regulares que são possivelmente tipos infinitos que são desmembramentos de árvores finitas. Por exemplo, isso torna possível uma definição do combinador "Y" de "Curry":

```
dec Ycurry : (alpha -> alpha) -> alpha;
--- Ycurry f <= Z Z where Z == lambda z => f(z z);
```

Isso falha no sistema de tipo usual porque "Z" deve ter um tipo de função com tipo de argumento igual ao tipo inteiro.

Os sinônimos de tipo podem ser recursivos para que possam ser usados para descrever tipos infinitos como o tipo de sequências infinitas:

```
type seq alpha == alpha # seq alpha;
```

Os usos recursivos do construtor de tipo que está sendo definido deve ter exatamente os mesmos argumentos do lado esquerdo. No entanto, esses argumentos podem ser permutados como em:

```
type alternate alpha beta == alpha # alternate beta alpha;
```

A sintaxe de tipos também é estendida para expressar tipos regulares. Por exemplo, os dois tipos acima podem ser definidos como:

```
type seq alpha == mu s => alpha # s;
type alternate alpha beta == mu a => alpha # (beta # a);
```

Essa notação também é usada pelo sistema para imprimir tipos regulares.

Functores

Cada "dado" ou "definição de tipo" introduz uma nova função "map" '(ou "functor" para os teóricos da categoria) com o mesmo nome e aridade (quantidade de argumentos de uma função[7]) que o construtor de "tipo". Por exemplo, a definição de tipo

```
data tree alpha == Empty ++ Node (tree alpha) alpha (tree alpha);
```

também define uma função

```
tree : (alpha -> beta) -> tree alpha -> tree beta
```

que mapeia uma função em árvores. Esta definição automática pode ser substituída explicitamente.

É um pouco mais complicado: se o argumento de tipo for usado negativamente, como em

```
type cond alpha == alpha -> bool;
```

a função terá um tipo invertido:

```
cond : (alpha -> beta) -> tree beta -> tree alpha
```

Se o argumento for usado positivamente e negativamente, como em

```
type endo alpha == alpha -> alpha;
```

a função terá um tipo como

```
endo : (alpha -> alpha) -> tree alpha -> tree alpha
```

Da mesma forma uma definição de "abstype" declara esta função correspondente. Para determinar o tipo, cada argumento é assumido como sendo usado positivamente e negativamente a menos que a variável do argumento seja substituída por uma das palavras chave especiais "pos", "neg" ou "none" (indicando que o argumento não é usado). Veja o apêndice anterior para alguns exemplos.

BIBLIOGRAFIA

1 Roger Bailey.
A Hope tutorial.
Byte, pages 235-258, August 1985.

[7] Nota do tradutor.

2 Roger Bailey.
Functional Programming in Hope.
Ellis Horwood, Chichester, England, 1990.

3 R.M. Burstall, D.B MacQueen, and D.T. Sanella.
Hope: An experimental applicative language.
In The 1980 LISP Conference, pages 136-143, Stanford, August 1980.
Also CSR-62-80, Dept of Computer Science, University of Edinburgh.

4 A.J. Field and P.G. Harrison.
Functional Programming.
Addison Wesley, Wokingham, England, 1988.

ÍNDICE REMISSIVO