Sandra Maria Mendes Alves

Inferência de Tipos e Resolução de Restrições

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto

2001

Sandra Maria Mendes Alves

Inferência de Tipos e Resolução de Restrições

Tese submetida à Faculdade de Ciências da Universidade do Porto para obtenção do grau de Mestre em Informática

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2001

Aos meus pais e aos meus irmãos...

Agradecimentos

Em primeiro lugar gostaria de agradecer ao meu orientador, Mário Florido, pelo apoio que me deu ao longo do desenvolvimento deste trabalho. Em particular gostaria de agradecer a constante disponibilidade, o incentivo, os comentários pertinentes, a minúcia com que leu esta dissertação, e na fase final o sacrifício pessoal feito para que o trabalho fosse terminado na data pretendida.

Aos meus colegas do LIACC, pelo óptimo ambiente de trabalho, pela amizade e ajuda, e pelos almoços sempre divertidos. Para não correr o risco de me esquecer de alguém, agradeço de um modo geral a todos os que me ajudaram, quer com pequenas dúvidas acerca de LaTeX, quer mostrando interesse e preocupação com o meu trabalho. Não posso deixar de agradecer de uma forma especial ao meu colega de gabinete Luís Antunes, pela sua amizade e boa companhia.

Agradeço a Martin Odersky e a Martin Sulzmann por responderem sempre com prontidão às minhas perguntas e pelos seus comentários elucidativos que me ajudaram a compreender melhor alguns aspectos do HM($\mathcal{X}$).

Ao meu amigo Nuno Fonseca, por ter lido a versão final da tese em tão pouco tempo.

Não posso deixar de agradecer de forma especial à minha família, em especial aos meus pais que sempre me apoiaram em todas as minhas decisões.

Finalmente, um agradecimento especial ao Rui por todo o apoio e compreensão nos últimos tempos.

Índice

Resumo 13

1 Introdução 15

1.1 Trabalho relacionado 17

1.2 Resumo da dissertação 17

2 Preliminares 19

2.1 Sistemas de tipos para o λ-calculus 19

2.1.1 O λ-calculus 19

2.1.1.1 Definição e sintaxe 19

2.1.1.2 Variáveis e Substituições 20

2.1.1.3 Reduções e Forma Normal 22

2.1.2 O Sistema de Tipos Simples 23

2.1.2.1 Tipos 23

2.1.2.2 Regras de Inferência 24

2.1.2.3 Substituições 25

2.1.2.4 Tipos Principais 26

2.1.2.5 Unificação 26

2.1.2.6 Inferência de tipos 28

2.1.3 O Sistema de Tipos de Damas-Milner 30

2.1.3.1 Polimorfismo paramétrico 30

2.1.3.2 A linguagem de termos 30

2.1.3.3 Esquemas de tipos 31

2.1.3.4 Tipos 31

2.1.3.5 O Sistema de tipos 31

2.1.3.6 Inferência de Tipos 32

2.2 CHR *(Constraint Handling Rules)* 34

2.2.1 Programação por Restrições 34

2.2.1.1 Restrições 35

2.2.1.2 Resolução de Restrições 36

2.2.2 CHR 37

2.2.2.1 Restrições 37

2.2.2.2 Sintaxe 37

2.2.2.3 Semântica Declarativa 38

2.2.2.4 Semântica Operacional 38

2.2.2.5 Confluência 40

3 Inferência de Tipos e Resolução de Restrições **43**

3.1 HM($\mathcal{X}$) 44

3.1.1 Sistemas de Restrições 44

3.1.1.1 Sistemas de Restrições Simples 44

3.1.1.2 Sistemas de Restrições Cilíndricos 44

3.1.1.3 Sistemas de Restrições sobre Termos 45

3.1.2 Os Sistemas de Tipos HM($\mathcal{X}$) 46

3.1.2.1 A linguagem de termos 47

3.1.2.2 A linguagem de tipos 47

3.1.2.3 Regras de Inferência 47

3.1.3 Inferência de Tipos em HM($\mathcal{X}$) 49

3.1.3.1 Normalização 50

3.1.3.2 Algoritmo 51

3.1.3.3 Integridade e Completude 53

3.1.4 Tipos Principais 55

3.2 Inferência de Tipos e Programação por Restrições 55

3.3 HM($\mathcal{DM}$) 57

3.3.1 O sistema de restrições $\mathcal{DM}$ 57

3.3.2 A linguagem de tipos $\mathcal{T}$ 57

3.3.3 O conjunto de restrições em forma resolvida $\mathcal{S}$ 58

3.3.4 O ambiente inicial Γ_0 58

3.3.5 Exemplos 58

4 Estruturas com Campos **61**

4.1 O sistema $\mathcal{O}$ de Ohori 61

4.1.1 Linguagem de termos 61

4.1.2 Tipos e Espécies 62

4.1.3 Regras de inferência 66

4.2 O HM($\mathcal{R}$) 68

4.2.1 A linguagem de tipos $\mathcal{T}$ 68

4.2.2 A linguagem de termos e o ambiente inicial Γ_0 69

4.2.3 O sistema de restrições $\mathcal{R}$ 70

4.2.4 O conjunto $\mathcal{S}$ 72

4.2.5 Exemplos 73

5 Sobrecarga **75**

5.1 Classes de Tipos 75

5.1.1 O Sistema com Classes de Nipkow e Prehofer 76

5.1.1.1 Tipos e Classes 76

5.1.1.2 Classes e Instâncias 77

5.1.1.3 Equivalência entre Conjuntos de Classes 78

5.1.1.4 Linguagem de termos 79

5.1.1.5 Regras de Inferência 80

5.1.1.6 As classes do Haskell e o sistema de Nipkow e Prehofer . . . 84

5.1.2 Mundo Aberto *versus* Mundo Fechado 84

5.1.3 O HM($\mathcal{TC}$) 85

5.1.3.1 A linguagem de tipos $\mathcal{T}$ 85

5.1.3.2 A linguagem de termos e o ambiente inicial Γ_0 86

5.1.3.3 O sistema de restrições $\mathcal{TC}$ 86

5.1.3.4 O conjunto de restrições em forma resolvida $\mathcal{S}$ 88

5.1.4 Exemplos 88

5.1.5 Ambiguidade 90

5.2 Sobrecarga de operadores e o Sistema $\mathcal{O}$ 91

5.2.1 Sistema $\mathcal{O}$ 91

5.2.1.1 Tipos 92

5.2.1.2 A linguagem de termos do sistema $\mathcal{O}$ 92

5.2.1.3 Regras de Inferência 95

5.2.2 O HM($\mathcal{O}$) 96
5.2.2.1 A linguagem de tipos $\mathcal{T}$ 96
5.2.2.2 O ambiente inicial Γ_0 97
5.2.2.3 O sistema de restrições $\mathcal{O}$ 97
5.2.2.4 O conjunto de restrições em forma resolvida $\mathcal{S}$ 98
5.2.2.5 Exemplos 99

6 Conclusão **101**
6.1 Trabalho futuro 102
6.2 Considerações Finais 102

Referências **102**

A HM($\mathcal{X}$) em Prolog **107**
A.1 Algoritmo de inferência para o HM($\mathcal{X}$) em Prolog 107
A.2 Módulos em CHR 111
A.2.1 Sistema com estruturas com campos 111
A.2.2 Sistema com classes de tipos 112
A.2.3 Sistema $\mathcal{O}$ 112

Resumo

Nesta dissertação apresentamos uma implementação do esquema geral de algoritmos de inferência de tipos HM($\mathcal{X}$), usando Prolog e CHR (Constraint Handling Rules).

O processo de inferência de tipos pode envolver mecanismos de resolução de restrições. Embora as restrições de igualdade possam ser resolvidas por um algoritmo de unificação, outros tipos de restrições podem necessitar de algoritmos mais complexos. Na nossa implementação, as restrições de igualdade entre tipos são resolvidas por unificação, usando a unificação do Prolog com *occur-check*, e outros tipos de restrições são resolvidas por resolutores específicos implementados em CHR.

Com base no esquema geral de algoritmos de inferência de tipos HM($\mathcal{X}$), implementamos:

1. Um algoritmo geral de inferência de tipos;
2. Instâncias para:
   (a) o sistema de Damas-Milner;
   (b) o sistema com estruturas com campos de Ohori;
   (c) o sistema de Nipkow e Prehofer com classes de tipos;
   (d) o sistema $\mathcal{O}$ de Odersky, Wadler e Wehr.

Na nossa implementação, a diferença entre os aspectos gerais dos algoritmos de inferência de tipos e os módulos de resolução de restrições torna-se mais clara quando comparada com outras implementações dos mesmos sistemas de tipos, normalmente feitas numa linguagem de programação funcional. As regras CHR revelaram-se um modo natural de especificar os mecanismos de simplificação.

Capítulo 1

Introdução

O conceito de restrições em sistemas de tipos não é novo. Muitos sistemas de tipos usam mecanismos de resolução de restrições no processo de inferência. Por exemplo, o algoritmo de inferência de tipos da linguagem ML [DM82] usa unificação para resolver restrições de igualdade na linguagem de tipos. Ao unificarmos um tipo τ_1 com um tipo τ_2 estamos a impor uma restrição de igualdade $\tau_1 = \tau_2$ entre os tipos τ_1 e τ_2, sendo o algoritmo de unificação o método de resolução destas restrições.

Nos últimos anos, apareceram várias extensões ao sistema de tipos Damas-Milner, algumas delas baseadas em resolução de restrições. Alguns exemplos incluem sistemas com estruturas com campos [Oho95], e sistemas de inferência de tipos para linguagens que lidam com sobrecarga de operadores [WB89, NP93, OWW95].

Em [SOW99], Odersky, Sulzmann e Wehr definiram o HM($\mathcal{X}$) como um esquema geral para inferência de tipos com restrições. A sua principal motivação prende-se com o facto de, apesar dos sistemas de tipos com restrições usarem domínios de restrições diferentes, eles serem semelhantes nos seus fundamentos teóricos.

O HM($\mathcal{X}$) é parametrizado com um sistema de restrições $\mathcal{X}$ que, instanciado a um sistema de restrições específico, define um sistema de tipos em particular (análogo ao que acontece com o esquema CLP($\mathcal{X}$) da programação em lógica por restrições [JL87], onde a instanciação do parâmetro $\mathcal{X}$ define uma linguagem de programação específica).

Em [SOW99] demonstrou-se que as instâncias de HM($\mathcal{X}$) são íntegras com respeito a uma semântica não tipada, e foi apresentado um algoritmo geral de inferência de tipos que garante propriedades comuns, tais como a decidibilidade e a existência de tipos principais, mediante certas condições sobre $\mathcal{X}$.

A nossa principal motivação inicial foi verificar até que ponto a inferência de tipos baseada em restrições se torna mais clara quando implementada numa linguagem de programação por restrições.

Dada a grande diversidade de domínios de restrições para os diferentes sistemas de tipos, a escolha de Constraint Handling Rules (CHR) [Fru98], (que permite facilmente implementar resolutores específicos), para especificar a resolução das restrições parece-nos natural.

Vamos usar o seguinte exemplo para ilustrar o género de problemas que nos propusemos a resolver. Consideremos o seguinte programa em Haskell:

$$\begin{array}{l}
\textbf{class}\ Eq\,\alpha\ \ \textbf{where}\ eq\ :\ \alpha\ \rightarrow\ \alpha\ \rightarrow\ bool \\
\textbf{class}\ Eq\,\alpha\ \Rightarrow\ Ord\,\alpha\,\textbf{where}\ le\ :\ \alpha\ \rightarrow\ \alpha\ \rightarrow\ bool \\
g\ x\ y\ =\ le\ x\ y
\end{array}$$

Na inferência do tipo de g temos:

$$g\ ::\ \forall\,\alpha.\,(\alpha\ :\ Eq)\ \wedge\ (\alpha\ :\ Ord)\ \Rightarrow\ \alpha\ \rightarrow\ \alpha\ \rightarrow\ bool$$

O processo de inferência de tipo para a função g, gera restrições de igualdade entre tipos, que são resolvidas por unificação. Por exemplo, sendo o tipo inferido para le igual a $\alpha \rightarrow \alpha \rightarrow bool$, e o tipo inferido para x igual a β, então para inferir o tipo τ para $(le\ x)$ é necessário resolver por unificação a restrição de igualdade $\alpha \rightarrow \alpha \rightarrow bool = \beta \rightarrow \tau$, resultando em $\tau = \alpha \rightarrow bool$.

Também são geradas restrições de classes da forma $(\alpha : C)$, que são resolvidas por um módulo CHR que implementa a hierarquia de classes do programa. No exemplo acima, a restrição $(\alpha : Ord)$, considerando que Ord é subclasse de Eq, gera a restrição $(\alpha : Eq)$.

A escolha da linguagem Prolog com um resolutor de restrições específico para as classes, permite usar unificação para resolver restrições de igualdade, e o resolutor de restrições (implementado em CHR), para resolver as restrições de classes.

Nesta dissertação redefinimos o algoritmo geral para o HM($\mathcal{X}$) usando Prolog com CHR. O nosso objectivo não é apresentar mais um algoritmo de inferência de tipos. A principal contribuição deste trabalho, é mostrar que a programação em lógica, como paradigma de especificação de algoritmos de inferências de tipos, e o CHR como forma de especificar algoritmos de resolução de restrições implicitamente definidos pelos sistemas de tipos, são uma forma clara para facilmente especificar e implementar algoritmos de inferência de tipos para linguagens de programação.

1.1 Trabalho relacionado

Tanto quanto sabemos, o único trabalho existente que relaciona CHR com inferência de tipos para programação funcional foi exposto em [GSM00]. Neste artigo Glynn, Stuckey e Sulzmann aplicaram CHR ao estudo e implementação de várias extensões às classes de tipos. No nosso trabalho aplicamos CHR para especificação de qualquer sistema de inferência de tipos baseado em extensões à unificação dentro do esquema HM($\mathcal{X}$).

A especificação das classes de tipos em [GSM00] pode ser vista como um módulo CHR de simplificação para classes de tipos, que pode ser facilmente integrado na nossa implementação, permitindo inferência de tipos com sobrecarga de operadores usando um formalismo baseado em classes de tipos.

Como pré-requisitos, é esperado que o leitor esteja familiarizado com o paradigma da programação em lógica, em particular com a linguagem Prolog [SS94], e com linguagens de programação funcional (ver por exemplo [Tho96] como referência para a linguagem de programação Haskell).

1.2 Resumo da dissertação

No segundo capítulo da dissertação introduzimos conceitos e resultados básicos importantes para o trabalho efectuado, nomeadamente relacionados com sistemas de inferência de tipos para o $\lambda-$calculus, e programação em lógica por restrições.

Em particular, descrevemos o sistema de inferência de tipos de Damas-Milner, em que são baseados os sistemas de tipos que descreveremos ao longo da tese, e o CHR como linguagem de implementação de resolutores específicos de restrições.

No terceiro capítulo é apresentada a nossa implementação do algoritmo de inferência de tipos para o HM($\mathcal{X}$). Começamos por descrever o HM($\mathcal{X}$). Falamos de sistemas de restrições que definem o parâmetro $\mathcal{X}$, de normalização de restrições como processo de resolução, e descrevemos o algoritmo geral de inferência de tipos para o HM($\mathcal{X}$).

Com base nesse algoritmo, apresentamos uma implementação em Prolog, que usa CHR na especificação das regras de normalização das restrições para cada sistema de restrições. Como primeiro caso de estudo, apresentamos o sistema Damas-Milner como instância do HM($\mathcal{X}$).

No capítulo quatro apresentamos o sistema de tipos de Ohori [Oho95]. O sistema de tipos de Ohori, extende o sistema de tipos de Damas-Milner com estruturas com campos. Apresentamos o sistema de Ohori como uma instância do HM($\mathcal{X}$), e definimos as regras CHR de normalização de restrições neste sistema.

No capítulo seguinte descrevemos dois sistemas que lidam com sobrecarga de operadores: o sistema de classes de tipos de Nipkow e Prehofer [NP93], e o sistema $\mathcal{O}$ de Odersky, Wadler e Wehr [OWW95]. O sistema de classes de Nipkow e Prehofer apresenta um formalismo de classes de tipos tal como o utilizado na linguagem Haskell. O sistema $\mathcal{O}$ apresenta uma forma restrita de sobrecarga de operadores que evita o problema de ambiguidade na inferência de tipos.

Para cada um dos sistemas apresentados, definimos o sistema de restrições correspondente, e apresentamos um módulo em CHR de resolução de restrições.

No último capítulo da dissertação apresentamos algumas conclusões finais e apontamos algumas linhas futuras de investigação.

Apresentamos em apêndice, o código da implementação do algoritmo em Prolog, e os módulos de resolução de restrições em CHR para os sistemas de tipos estudados.

O trabalho desenvolvido nesta dissertação foi parcialmente exposto em [AF01].

Capítulo 2

Preliminares

Ao longo deste capítulo vamos apresentar definições e resultados importantes para o trabalho apresentado nos capítulos seguintes.

Em primeiro lugar, vamos falar de sistemas de inferência de tipos para o λ-calculus, que estão na base dos sistemas de tipos de que falaremos mais adiante.

A segunda parte deste capítulo pretende introduzir alguns conceitos básicos de programação em lógica por restrições, e do CHR como linguagem de escrita de resolutores.

2.1 Sistemas de tipos para o λ-calculus

2.1.1 O λ-calculus

Em 1933, Alonzo Church [Chu33] definiu o λ-calculus. Mais tarde, este viria a tornar-se útil como modelo formal de computação.

Vamos introduzir alguns conceitos básicos relativos ao λ-calculus necessários para a continuação do nosso trabalho. Para um estudo mais aprofundado do assunto ver [Bar84].

2.1.1.1 Definição e sintaxe

Definição 2.1.1 *Seja $\mathcal{V}$ um conjunto infinito de variáveis. O conjunto de λ-termos, Λ é construído a partir do conjunto $\mathcal{V}$ usando aplicação e abstracção da seguinte forma:*

$$
\begin{array}{lrcl}
 & x \in \mathcal{V} & \Rightarrow & x \in \Lambda \\
(Aplica\c{c}\tilde{a}o) & M, N \in \Lambda & \Rightarrow & (MN) \in \Lambda \\
(Abstrac\c{c}\tilde{a}o) & M \in \Lambda,\, x \in \mathcal{V} & \Rightarrow & (\lambda x.M) \in \Lambda
\end{array}
$$

A aplicação (MN) representa a aplicação da função representada por M a um argumento N. A abstracção $(\lambda x.M)$ representa a função f tal que $f(x) = M$. Aplicar f a N tem como resultado a substituição de x por N em M.

Notação: Escrevemos $M \equiv N$ para indicar igualdade sintática entre termos. Considerando ainda que a aplicação é associativa à esquerda, e a abstracção associativa à direita, podemos utilizar as seguintes abreviações, de forma a não sobrecarregar a notação:

- $(M_1 M_2 \ldots M_n)$ para $(\ldots(M_1 M_2)\ldots M_n)$
- $(\lambda x_1 x_2 \ldots x_n.M)$ para $(\lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.M)\ldots)))$

Assim, o termo $(\lambda x.(\lambda z.((xz)(\lambda y.(yx)))))$, pode ser escrito como $(\lambda xz.xz(\lambda y.yx))$.

2.1.1.2 Variáveis e Substituições

Vamos diferenciar o tipo de ocorrências das variáveis, formalizando os conceitos de variáveis *livres* e *ligadas* de um termo.

Definição 2.1.2 *Dado $M \in \Lambda$, o conjunto $FV(M)$ das variáveis livres de M é definido indutivamente por:*

$$
\begin{array}{rcl}
FV(x) & = & \{x\} \\
FV(MN) & = & F(M) \cup FV(N) \\
FV(\lambda x.M) & = & FV(M) \setminus \{x\}
\end{array}
$$

Definição 2.1.3 *Dado $M \in \Lambda$, o conjunto $BV(M)$ das variáveis ligadas de M é definido indutivamente por:*

$$
\begin{array}{rcl}
BV(x) & = & \emptyset \\
BV(MN) & = & BV(M) \cup BV(N) \\
BV(\lambda x.M) & = & BV(M) \cup \{x\}
\end{array}
$$

Um termo diz-se fechado se e só se $FV(M) = \emptyset$. O conjunto de todos os λ-termos *fechados* designa-se por $\Lambda^0 \subset \Lambda$.

Por exemplo, no termo $(\lambda zx.yx)$, a variável x ocorre *ligada* e a variável y ocorre *livre*. Note-se que os conjuntos de variáveis *livres* e *ligadas* de um termo não são necessariamente disjuntos; x ocorre *livre* e *ligada* em $x(\lambda xy.x)$.

Definição 2.1.4 (Substituição) *$M[L/y]$, o resultado de substituir todas as ocorrências livres de y em M por L é dado por:*

$$x[N/y] \equiv \begin{cases} L & se\ x \equiv y \\ x & caso\ contrário \end{cases}$$

$$(MN)[L/y] \equiv (M[L/y]N[L/y])$$

$$(\lambda x.M)[L/y] \equiv \begin{cases} (\lambda x.M) & se\ x \equiv y \\ (\lambda x.M[L/y]) & caso\ contrário \end{cases}$$

Um problema que pode surgir com a substituição é a *captura de variáveis*. Consideremos o termo $(\lambda x.(\lambda y.x))$, representando a função que, quando aplicada a um argumento N, retorna a função constante $(\lambda y.N)$. Se $N \equiv y$, obtemos $(\lambda y.x)[y/x] \equiv (\lambda y.y)$. Ou seja, substituindo x por y obtemos a função identidade, pois a ocorrência livre de x torna-se numa ocorrência ligada de y.

Assim, a substituição $M[N/x]$ só é permitida, e nesse caso dizemos que x é substituível por N em M, se as variáveis ligadas de M forem disjuntas das variáveis livres de N:

$$BV(M) \cap FV(N) = \emptyset$$

Para garantir que esta condição se verifica podemos, se necessário, renomear as variáveis ligadas de M. A essa mudança de variáveis ligadas chama-se conversão-α.

Definição 2.1.5 *Uma mudança de uma variável ligada x num termo M é a substituição de todos os sub-termos de M da forma $(\lambda x.N)$ por $(\lambda y.N[y/x])$, onde y é uma variável que não ocorre em N.*

A mudança de variáveis ligadas preserva o significado do termo, (no sentido em que representa a mesma função). A essa noção chamamos *congruência-α*:

Definição 2.1.6 (Congruência-α) *M é α-congruente com N, (notação $M \equiv_\alpha N$), se N pode ser obtido de M por uma série de mudanças de variáveis ligadas, e vice-versa.*

Por exemplo:

$$\lambda x.xy \equiv_\alpha \lambda z.zy \not\equiv_\alpha \lambda y.yy$$

Daqui em diante assumimos que os conjuntos de variáveis livres e ligadas são sempre disjuntos. Assim, qualquer substituição $M[N/x]$ é válida.

2.1.1.3 Reduções e Forma Normal

Definição 2.1.7 (Redução-β) *Um λ-termo M reduz-se a N por um passo de redução-β, e escreve-se $M \rightarrow_{1\beta} N$, se N pode ser obtido substituindo em M um sub-termo da forma $(\lambda x.P)Q$ por $P[Q/x]$. Definimos a relação $\twoheadrightarrow_\beta$ como sendo o fecho reflexivo e transitivo de $\rightarrow_{1\beta}$.*

Chama-se *β-redex* a um λ-termo da forma $(\lambda x.P)Q$ sendo $P[Q/x]$ o seu *contractum*.

Definição 2.1.8 *Um termo que não contém qualquer β-redex, não admite mais reduções-β e diz-se em forma normal-β ou simplesmente em forma normal.*

Dizemos que um λ-termo M admite forma normal se pode ser reduzido a um termo N em forma normal, $M \twoheadrightarrow_\beta N$. O seguinte teorema garante que não existem duas sequências de redução, partindo de um mesmo λ-termo, que terminem em duas formas normais distintas. Ou seja, se um termo admite forma normal, esta é única.

Teorema 2.1.1 (Church-Rosser) *Se $M \twoheadrightarrow N_1$ e $M \twoheadrightarrow N_2$ então existe N tal que $N_1 \twoheadrightarrow N$ e $N_2 \twoheadrightarrow N$.*

Note-se que não consideramos distintos dois λ-termos α-congruentes. Ou seja, se $M \twoheadrightarrow N_1$ e $M \twoheadrightarrow N_2$ e N_1, N_2 estão em forma normal então $N_1 \equiv_\alpha N_2$.

Vimos que um termo não pode ser reduzido a duas formas normais distintas. No entanto, nem todas as sequências de redução levam à forma normal do termo. Consideremos o λ-termo $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$. Ao tentarmos reduzir Ω obtemos novamente o próprio Ω, o que leva à não terminação do processo de redução.

Consideremos ainda o termo $(\lambda xy.y)\Omega$. A redução

$$\underline{(\lambda xy.y)\Omega} \rightarrow \lambda y.y$$

termina com a forma normal do termo, eliminando Ω. No entanto, se tentarmos normalizar o argumento Ω, obtemos a seguinte sequência de redução:

$$(\lambda xy.y)\underline{\Omega} \to (\lambda xy.y)\underline{\Omega} \to \cdots$$

Um λ-termo M com forma normal, que admita uma sequência infinita de redução, atinge essa forma normal, se os sub-termos de M que não tenham forma normal forem eliminados por uma redução.

Definição 2.1.9 (Redução em ordem normal) *Uma sequência de reduções diz-se em ordem normal se, em cada passo, escolhermos o β-redex mais à esquerda no λ-termo.*

Teorema 2.1.2 *Se M admite forma normal N, então existe uma sequência de reduções em ordem normal a partir de M que termina em N.*

Se pensarmos na redução em termos de computação então a forma normal é o resultado da computação, quando esta termina. A redução em ordem normal corresponde à *chamada por nome* na avaliação de funções.

2.1.2 O Sistema de Tipos Simples

O sistema de tipos simples [Cur34] permite determinar o carácter funcional (domínio e contradomínio) de alguns λ-termos.

Este sistema está na base dos sistemas de tipos para linguagens de programação tipificadas, como o ML e o Haskell.

2.1.2.1 Tipos

Partindo de um conjunto infinito de variáveis de tipo $\mathbb{V}$, o tipo $(\alpha \to \beta)$ representa as funções que, quando aplicadas a elementos de tipo α, têm como resultado um elemento de tipo β. O conjunto $\mathbb{T}$ de tipos é construído da seguinte forma:

$$\begin{array}{rcl} \alpha \in \mathbb{V} & \Rightarrow & \alpha \in \mathbb{T} \\ \tau, \tau' \in \mathbb{T} & \Rightarrow & (\tau \to \tau') \in \mathbb{T} \end{array}$$

Notação: Se $\sigma_1, \ldots, \sigma_n \in \mathbb{T}$, então

$$\sigma_1 \to \sigma_2 \to \cdots \to \sigma_n$$

representa

$$(\sigma_1 \to (\sigma_2 \to \cdots \to (\sigma_{n-1} \to \sigma_n) \dots));$$

ou seja, os tipos são associativos à direita.

Assim, o tipo $(\alpha \to \beta \to \gamma)$, representa as funções que, quando aplicadas a elementos do tipo α retornam uma função do tipo $(\beta \to \gamma)$, e o tipo $((\alpha \to \beta) \to \gamma)$, representa as funções que, quando recebem uma função do tipo $(\alpha \to \beta)$ como argumento, têm como resultado um elemento do tipo γ.

Definição 2.1.10 *Dados $x \in \mathcal{V}$ e $\tau \in \mathbb{T}$ dizemos que:*

- *Uma declaração é um par da forma $x : \tau$ onde o tipo τ é o predicado e a variável x é o sujeito da declaração.*
- *Uma base Γ é um conjunto de declarações com variáveis distintas como sujeitos.*
- *O domínio de uma base $\Gamma = \{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$, $dom(\Gamma)$, é $\{x_1, \dots, x_n\}$, e $\sigma_i = \Gamma(x_i)$.*
- *Definimos Γ_x como sendo $\Gamma \setminus \{x : \tau\}$, onde Γ é uma base.*

2.1.2.2 Regras de Inferência

No sistema de tipos simples uma tipagem $M : \sigma$ é derivada duma base Γ e escrevemos $\Gamma \vdash M : \sigma$, se $\Gamma \vdash M : \sigma$ pode ser obtida pela aplicação das seguintes regras de inferência:

$$\begin{array}{lrcl}
(\mathit{Axioma}) & (x : \sigma) \in \Gamma & \Rightarrow & \Gamma \vdash x : \sigma \\
(\mathit{Aplicação}) & \Gamma \vdash M : (\sigma \to \tau) \wedge \Gamma \vdash N : \sigma & \Rightarrow & \Gamma \vdash (MN) : \tau \\
(\mathit{Abstracção}) & \Gamma \cup \{x : \sigma\} \vdash M : \tau & \Rightarrow & \Gamma \vdash (\lambda x.M) : (\sigma \to \tau)
\end{array}$$

A notação normalmente usada para as regras descritas acima é a seguinte:

$$(\mathit{Axioma}) \qquad \Gamma \vdash x : \sigma, \text{ se } (x : \sigma) \in \Gamma$$

$$(\mathit{Aplicação}) \qquad \frac{\Gamma \vdash M : (\sigma \to \tau),\ \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau}$$

$$(\mathit{Abstracção}) \qquad \frac{\Gamma \cup \{x : \sigma\} \vdash M : \tau}{\Gamma \vdash (\lambda x.M) : (\sigma \to \tau)}$$

Exemplo 2.1.1 *Como exemplo de uma derivação de tipos no sistema acima descrito temos a seguinte árvore de derivação para o termo* $(\lambda xy.x)(\lambda x.x)$*.*

$$\begin{tikzcd}
\{x:\alpha\to\alpha, y:\beta\}\vdash x:\alpha\to\alpha \arrow[d, swap, "ABS"] & & \\
\{x:\alpha\to\alpha\}\vdash \lambda y.x:\beta\to\alpha\to\alpha \arrow[d, swap, "ABS"] & & \{x:\alpha\}\vdash x:\alpha \arrow[d, swap, "ABS"] \\
\vdash \lambda xy.x:(\alpha\to\alpha)\to\beta\to\alpha\to\alpha \arrow[dr, swap, "APP"] & & \vdash \lambda x.x:\alpha\to\alpha \arrow[dl, "APP"] \\
& \vdash (\lambda xy.x)(\lambda x.x):\beta\to\alpha\to\alpha &
\end{tikzcd}$$

2.1.2.3 Substituições

Uma vez que as variáveis de tipo podem ser substituídas por tipos vamos formalizar a definição de substituição:

Definição 2.1.11 (Substituição) *Consideremos um conjunto* $\mathbb{V}$ *de variáveis de tipos e um conjunto* $\mathbb{T}$ *de tipos. Uma substituição* $S = [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ *é uma função idempotente (ou seja,* $\forall\sigma[S(S\sigma) = S\sigma]$*)* $S : \mathbb{V} \to \mathbb{T}$ *que transforma cada* α_i *em* τ_i *e todas as outras variáveis nelas próprias.*

Definição 2.1.12 (Composição) *Sejam* S_1 *e* S_2 *duas substituições. A composição de substituições* $S_2 \circ S_1$ *é definida como:*

$$\forall\alpha \in \mathbb{V} \quad (S_2 \circ S_1)(\alpha) = S_2(S_1(\alpha))$$

Com a definição de substituição podemos definir a noção de instância:

Definição 2.1.13 (Instância) *Dizemos que um tipo* τ' *é uma instância de um tipo* τ *se e só se existe uma substituição* S *tal que* $\tau' = S\tau$*.*

Analogamente definimos a noção de instância de uma base:

Definição 2.1.14 *Dizemos que uma base* Γ' *é uma instância de uma base* Γ *se e só se existe uma substituição* S *tal que* $\Gamma' = S\,\Gamma$*.*

2.1.2.4 Tipos Principais

Um termo $M \in \Lambda$ pode ter mais do que um tipo. Por exemplo, para o termo $(\lambda x.x)$ conseguimos derivar, no sistema de Curry, os tipos $(\alpha_1 \to \alpha_1)$ e $((\alpha_2 \to \alpha_2) \to (\alpha_2 \to \alpha_2))$.

Assim, a noção de tipo principal surge do facto de, para um dado termo M, existir um tipo de M que é mais geral do que todos os outros, no sentido de qualquer tipo de M ser uma instância do tipo principal.

No exemplo acima, $((\alpha_2 \to \alpha_2) \to (\alpha_2 \to \alpha_2))$ é uma instância de $(\alpha_1 \to \alpha_1)$, onde $S = [(\alpha_2 \to \alpha_2)/\alpha_1]$.

Definição 2.1.15 (Tipo Principal) *Seja $M \in \Lambda$ e $\pi \in \mathbb{T}$, π é o tipo principal de M, se e só se:*

1. *Existe P tal que $P \vdash M : \pi$*
2. *Para qualquer derivação $B \vdash M : \sigma$ há um subconjunto B' de B tal que B' é uma instância de P e σ é uma instância de π.*

Dado que o tipo principal de um termo representa o conjunto de tipos para o termo, estamos particularmente interessados em algoritmos que, para um dado termo, determinem o seu tipo principal.

2.1.2.5 Unificação

O algoritmo de unificação de Robinson [Rob65] tem um papel central na inferência de tipos. No nosso trabalho a unificação de tipos é feita usando a unificação do Prolog com um teste de ocorrência. No entanto, e dada a sua importância, vamos apresentar aqui uma descrição do algoritmo de unificação para o caso particular em que os termos a unificar são tipos, ou seja variáveis, ou termos com functor $\to$ de aridade 2.

Definição 2.1.16 (Unificador) *Um unificador entre dois tipos τ_1 e τ_2, é uma substituição S tal que $S\,\tau_1 = S\,\tau_2$. Se dois tipos têm um unificador, então dizemos que os tipos unificam. A $S\,\tau_1$ (ou $S\,\tau_2$), chamamos instância comum.*

Exemplo 2.1.2 *Os tipos $(\alpha \to \beta \to \alpha)$ e $((\gamma \to \gamma) \to \delta)$ unificam. Para a substituição $S = [(\gamma \to \gamma)/\alpha, \beta \to (\gamma \to \gamma)/\delta]$, a instância comum dos dois tipos é $((\gamma \to \gamma) \to \beta \to (\gamma \to \gamma))$.*

Definição 2.1.17 (Unificador mais geral) *S é unificador mais geral de dois tipos τ_1 e τ_2 se, qualquer outro unificador de τ_1 e τ_2, S_1, existe uma substituição S_2 tal que $S_1 = S_2 \circ S$.*

Exemplo 2.1.3 *Consideremos os tipos $\tau_1 = (\alpha \to \alpha)$ e $\tau_2 = (\beta \to \gamma)$. A substituição $S' = [(\alpha_1 \to \alpha_2)/\alpha, (\alpha_1 \to \alpha_2)/\beta, (\alpha_1 \to \alpha_2)/\gamma]$ é um unificador τ_1 e τ_2, mas não é o unificador mais geral.*

O unificador mais geral de τ_1 e τ_2 é $S = [\alpha/\beta, \alpha/\gamma]$. A instância comum a τ_1 e τ_2 por S', $(\alpha_1 \to \alpha_2) \to (\alpha_1 \to \alpha_2)$ é uma instância de $(\alpha \to \alpha)$.

Vamos agora apresentar o algoritmo de Unificação de Robinson para o caso particular em que os termos a unificar são tipos. A função que vamos definir, quando aplicada a dois tipos τ_1 e τ_2, retorna o unificador mais geral de τ_1 e τ_2 caso exista e falha caso contrário.

Definição 2.1.18 (Algoritmo de Unificação) *Sejam τ_1 e τ_2 dois tipos. A função de unificação $UNIFY(\tau_1, \tau_2)$ é definida por:*

$$
\begin{array}{lcl}
UNIFY(\alpha, \tau) & = & [\tau/\alpha] \quad \textit{se } \alpha \notin FV(\tau) \\
 & = & Id \qquad \textit{(função identidade) se } \tau = \alpha \\
 & = & fail \quad \textit{caso contrário} \\
 & & \\
UNIFY(\tau_1 \to \tau_2, \alpha) & = & UNIFY(\alpha, \tau_1 \to \tau_2) \\
 & & \\
UNIFY(\sigma_1 \to \sigma_2, \tau_1 \to \tau_2) & = & \textit{let} \\
 & & \quad S = UNIFY(\sigma_2, \tau_2) \\
 & & \textit{in } \ UNIFY(S\,\sigma_1, S\,\tau_1) \circ S
\end{array}
$$

Considera-se que let $S = UNIFY(\sigma_2, \tau_2)$ in $UNIFY(S\,\sigma_1, S\,\tau_1) \circ S$ falha se uma das unificações falhar.

Na primeira linha, ao teste $\alpha \notin FV(\tau)$, chama-se teste de ocorrência. Este teste não existe por defeito na unificação do Prolog, por isso vamos usar nos próximos capítulos uma unificação do Prolog com teste de ocorrência.

2.1.2.6 Inferência de tipos

O sistema de tipos simples é decidível. Vamos descrever dois algoritmos baseados no algoritmo de unificação de Robinson [Rob65], que dada uma base Γ e um termo M determinam o tipo principal de M.

Uma vez que os algoritmos de inferência retornam o tipo principal de um dado termo M, para verificarmos que um dado tipo é derivável para o termo M basta verificar que é uma instância do termo principal.

Algoritmo de Hindley/Milner Inicialmente apresentado por Hindley em 1969 [Hin69], o algoritmo seguinte foi redescoberto por Milner em 1978 [Mil78], e define uma função que dado um termo M retorna uma base Γ e um tipo τ tal que $\Gamma \vdash M : \tau$, sendo τ o tipo principal de M.

Definição 2.1.19 *Seja Γ uma base, $M \in \Lambda$ e $\tau \in \mathbb{T}$. Seja UNIFY a unificação de Robinson. A função $T(M) = (\Gamma, \tau)$ define um algoritmo de inferência de tipos para o λ-calculus, da seguinte forma:*

1. *Se M é uma variável x então $\Gamma = \{x : \alpha\}$ e $\tau = \alpha$, sendo α uma variável nova;*
2. *Se $M \equiv M_1 M_2$, $T(M_1) = (\Gamma_1, \tau_1)$ e $T(M_2) = (\Gamma_2, \tau_2)$, então $T(M) = (S(\Gamma_1 \cup \Gamma_2), S\alpha)$, onde $S = UNIFY(\tau_1, \tau_2 \to \alpha)$ e α é uma variável nova;*
3. *Se $M \equiv \lambda x.N$ e $T(N) = (\Gamma', \tau_1)$ então:*
   a) *Se não há hipóteses para x em Γ', $T(M) = (\Gamma', \alpha \to \tau_1)$, onde α é uma variável nova;*
   b) *Se há uma hipótese $\{x : \tau_2\} \in \Gamma'$, $T(M) = (\Gamma'_x, \tau_2 \to \tau_1)$.*

Exemplo 2.1.4 *Consideremos a aplicação do algoritmo acima ao termo $(\lambda xy.x)(\lambda z.z)$.*

Temos:

$$\begin{array}{ll} T(x) = (\{x : \alpha\}, \alpha) & (por\ 1) \\ \Rightarrow T(\lambda y.x) = (\{x : \alpha\}, \beta \to \alpha) & (por\ 3.a) \\ \Rightarrow T(\lambda xy.x) = (\{\}, \alpha \to \beta \to \alpha) & (por\ 3.b) \end{array}$$

Também temos:

$$\begin{array}{ll} T(z) = (\{z : \gamma\}, \gamma) & (por\ 1) \\ \Rightarrow T(\lambda z.z) = (\{\}, \gamma \to \gamma) & (por\ 3.b) \end{array}$$

Dado que:

$$S = UNIFY(\alpha \to \beta \to \alpha, (\gamma \to \gamma) \to \delta) = [(\gamma \to \gamma)/\alpha, \beta \to (\gamma \to \gamma)/\delta]$$

Então :

$$T((\lambda xy.x)(\lambda z.z)) = (S(\{\}), S(\delta)) = (\{\}, \beta \to (\gamma \to \gamma)) \quad (por\ 2)$$

Algoritmo de Wand Este algoritmo [Wan87] dada uma base, um termo e um tipo retorna um conjunto de equações que, resolvidas usando unificação, determinam o tipo principal do termo.

Definição 2.1.20 *Seja Γ uma base, $M \in \Lambda$ e $\sigma \in \mathbb{T}$, tal que $FV(M) \subseteq dom(\Gamma)$. Consideremos $E = E(\Gamma, M, \sigma)$ o conjunto finito de equações definido da seguinte forma:*

$$\begin{array}{rcl} E(\Gamma, x, \sigma) & = & \{\sigma = \Gamma(x)\} \\ E(\Gamma, MN, \sigma) & = & E(\Gamma, M, \alpha \to \sigma) \cup E(\Gamma, N, \alpha), \\ & & \textit{onde } \alpha \textit{ é uma variável nova;} \\ E(\Gamma, \lambda x.M, \sigma) & = & E(\Gamma \cup \{x : \alpha\}, M, \beta) \cup \{\alpha \to \beta = \sigma\}, \\ & & \textit{onde } \alpha, \beta \textit{ são variáveis novas.} \end{array}$$

Aplicando a unificação ao conjunto E obtemos uma substituição S tal que $S(\Gamma) \vdash M : S(\sigma)$, e $S(\sigma)$ é o tipo principal de M.

Exemplo 2.1.5 *Exemplo de inferência de tipo para o termo $(\lambda xy.x)(\lambda x.x)$:*

$E(\{\}, (\lambda xy.x)(\lambda x.x), \sigma_0) =$
$E(\{\}, (\lambda xy.x), \alpha_1 \to \sigma_0) \cup E(\{\}, (\lambda x.x), \alpha_1)$

$E(\{\}, (\lambda xy.x), \alpha_1 \to \sigma_0) =$
$E(\{x : \alpha_2\}, (\lambda y.x), \beta_1) \cup \{\alpha_2 \to \beta_1 = \alpha_1 \to \sigma_0\} =$
$E(\{x : \alpha_2, y : \alpha_3\}, x, \beta_2) \cup \{\alpha_3 \to \beta_2 = \beta_1, \alpha_2 \to \beta_1 = \alpha_1 \to \sigma_0\} =$
$\{\beta_2 = \alpha_2, \alpha_3 \to \beta_2 = \beta_1, \alpha_2 \to \beta_1 = \alpha_1 \to \sigma_0\}$

$E(\{\}, (\lambda x.x), \alpha_1) =$
$E(\{x : \alpha_4\}, x, \beta_3) \cup \{\alpha_4 \to \beta_3 = \alpha_1\} =$
$\{\beta_3 = \alpha_4, \alpha_4 \to \beta_3 = \alpha_1\}$

$E = \{\beta_2 = \alpha_2, \alpha_3 \to \beta_2 = \beta_1, \alpha_2 \to \beta_1 = \alpha_1 \to \sigma_0, \beta_3 = \alpha_4, \alpha_4 \to \beta_3 = \alpha_1\}$

Aplicando o algoritmo de unificação ao conjunto E, obtemos:

$$\sigma_0 = \alpha_3 \to \alpha_4 \to \alpha_4$$

2.1.3 O Sistema de Tipos de Damas-Milner

2.1.3.1 Polimorfismo paramétrico

Consideremos o termo $(\lambda x.x)$, que representa a função identidade. No sistema de tipos simples podemos considerar as seguintes árvores de derivação de tipo para este termo:

$$\begin{array}{c} \{x : int\} \vdash x : int \\ \text{ABS}\big\downarrow \\ \vdash \lambda x.x : int \to int \end{array} \qquad\qquad \begin{array}{c} \{x : char\} \vdash x : char \\ \text{ABS}\big\downarrow \\ \vdash \lambda x.x : char \to char \end{array}$$

Em linguagens onde as funções ou procedimentos têm tipo único, não é possível definir funções que possam ser aplicadas a tipos de dados diferentes (como é o caso da função identidade), sendo necessário definir funções distintas para diferentes tipos. Tais linguagens, como por exemplo Pascal e C, são chamadas *monomórficas*. Linguagens com tipos polimórficos permitem definir tipos genéricos para objectos que tenham mais do que um tipo. Assim, acrescentando à linguagem de tipos descrições de conjuntos de tipos (esquemas de tipos), passamos a ter tipos polimórficos. Tais linguagens são chamadas *polimórficas*, e como exemplo temos o ML e o Haskell.

Existem vários tipos de polimorfismo. O polimorfismo da linguagem ML é o chamado *polimorfismo paramétrico*, onde as funções polimórficas funcionam uniformemente num conjunto de tipos.

No polimorfismo paramétrico, descrevemos conjuntos de tipos usando esquemas com parâmetros, que podem ser instanciados, permitindo uma representação do conjunto de tipos admissíveis para um determinado termo. No caso da função identidade $(\lambda x.x)$, o esquema de tipos que representa todos os tipos que podemos derivar para esta função é $(\forall\alpha.\alpha \to \alpha)$.

Muito brevemente, vamos falar de linguagens com sistemas de tipos com polimorfismo paramétrico. O sistema de tipos de Damas-Milner, que descreveremos em seguida, constitui a base dos sistemas de tipos para essas linguagens.

2.1.3.2 A linguagem de termos

O sistema de tipos de Damas-Milner [DM82], deu origem ao sistema de tipos da linguagem ML e foi construído para uma extensão ao λ-calculus.

Dado um conjunto infinito $\mathcal{V}$ de variáveis a linguagem de termos é definida pela seguinte gramática:

$$M ::= x \mid MM' \mid \lambda x.M \mid let\ x = M\ in\ M'$$

O termo $let\ x = M\ in\ M'$, apesar de semânticamente equivalente a $(\lambda x.M')M$, permite aumentar o conjuntos de termos tipificáveis. Isto porque, em $let\ x = M\ in\ M'$, x pode ser usado polimorficamente em M', apesar da expressão $(\lambda x.M')$ poder não ser tipificável. Veremos um exemplo da utilidade do *let* quando apresentarmos o sistema.

2.1.3.3 Esquemas de tipos

A definição de esquemas de tipos surge para, dados um termo M e uma base Γ, ser possível obter uma representação do conjunto de tipos que podemos inferir para M a partir de Γ.

Definição 2.1.21 *Dizemos que σ é um esquema de tipos se σ é um tipo τ ou é da forma $\forall \alpha_1, \ldots, \alpha_n.\tau$, onde $\alpha_1, \ldots, \alpha_n$ são variáveis de tipo chamadas variáveis genéricas.*

Definição 2.1.22 *Seja τ um tipo e σ um esquema de tipos, τ é uma instância genérica de σ se e só se $\sigma = \tau$ ou $\sigma = \forall \alpha_1, \ldots, \alpha_n.\tau'$ e $\exists \tau_1, \ldots, \tau_n$ tais que $\tau = [\tau_i/\alpha_i]\tau'$.*

Definição 2.1.23 *Dizemos que um esquema de tipos σ' é uma instância genérica de um esquema de tipos σ, se e só se todo o tipo τ que é instância genérica de σ', também é instância genérica de σ.*

2.1.3.4 Tipos

Definimos o conjunto de tipos deste sistema através da seguinte gramática:

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \tau' \to \tau'' \\ \sigma & ::= & \tau \mid \forall \alpha.\sigma' \end{array}$$

onde α pertence a um conjunto infinito de variáveis de tipo, τ, τ' e τ'' são tipos e σ e σ' são esquemas de tipos.

2.1.3.5 O Sistema de tipos

Consideremos $x \in \mathcal{V}$, $\alpha \in \mathbb{V}$, M e M' termos, τ e τ' tipos e σ e σ' esquemas de tipos. O sistema de tipos de Damas-Milner é definido pelas seguintes regras de inferência:

$$(Axioma) \qquad \Gamma \vdash x : \sigma, \text{ se } (x : \sigma) \in \Gamma$$

$$(Generalização) \qquad \frac{\Gamma \vdash M : \sigma, \alpha \text{ não ocorre livre em } \Gamma}{\Gamma \vdash M : \forall\alpha.\sigma}$$

$$(Instanciação) \qquad \frac{\Gamma \vdash M : \forall\alpha.\sigma}{\Gamma \vdash M : \sigma[\tau/\alpha]}$$

$$(Aplicação) \qquad \frac{\Gamma \vdash M : (\tau' \to \tau), \Gamma \vdash M' : \tau'}{\Gamma \vdash (MM') : \tau}$$

$$(Abstração) \qquad \frac{\Gamma \cup \{x : \tau'\} \vdash M : \tau}{\Gamma \vdash \lambda x.M : (\tau' \to \tau)}$$

$$(Let) \qquad \frac{\Gamma \vdash M : \sigma, \Gamma \cup \{x : \sigma\} \vdash M' : \tau}{\Gamma \vdash let\ x = M\ in\ M' : \tau}$$

Exemplo 2.1.6 *Consideremos a árvore de derivação de tipos para o termo let $y = (\lambda x.x)$ in yy:*

$$
\begin{tikzcd}
\{x : \alpha\} \vdash x : \alpha \arrow[d, "ABS"'] & & \{y : \forall\alpha.\alpha \to \alpha\} \vdash y : \forall\alpha.\alpha \to \alpha \arrow[d, "INS"'] & & \{y : \forall\alpha.\alpha \to \alpha\} \vdash y : \forall\alpha.\alpha \to \alpha \arrow[d, "INS"] \\
\vdash \lambda x.x : \alpha \to \alpha \arrow[d, "GEN"'] & & \{y : \forall\alpha.\alpha \to \alpha\} \vdash y : (\beta \to \beta) \to (\beta \to \beta) \arrow[dr, "APP"'] & & \{y : \forall\alpha.\alpha \to \alpha\} \vdash y : \beta \to \beta \arrow[dl, "APP"] \\
\vdash \lambda x.x : \forall\alpha.\alpha \to \alpha \arrow[drr, "LET"'] & & & \{y : \forall\alpha.\alpha \to \alpha\} \vdash yy : \beta \to \beta \arrow[dl, "LET"] & \\
& & \vdash let\ y = (\lambda x.x)\ in\ yy : \beta \to \beta & &
\end{tikzcd}
$$

Note-se que, o termo $(\lambda y.yy)(\lambda x.x)$, equivalente a let $y = (\lambda x.x)$ in yy, não admite tipo, uma vez que não é possível tipificar o subtermo $(\lambda y.yy)$.

2.1.3.6 Inferência de Tipos

Tal como o sistema de Curry, também este sistema é decidível. Uma noção necessária para descrever o algoritmo de inferência de tipos para este sistema é a noção de fecho de um tipo:

Definição 2.1.24 *Seja V um conjunto de variáveis, Γ uma base e $\tau \in \mathbb{T}$. O fecho de τ com respeito a V, $\overline{V}(\tau)$ é o esquema de tipos $\forall\alpha_1, \ldots, \alpha_n.\tau$, onde $\alpha_1, \ldots, \alpha_n$ são todas variáveis que ocorrem em τ e não pertencem a V. $\overline{\Gamma}(\tau)$ é o fecho de τ com respeito ao conjunto de variáveis que ocorrem em Γ.*

Uma vez definida a noção de fecho de um tipo, podemos descrever o algoritmo de inferência de tipos para o sistema de Damas-Milner.

Definição 2.1.25 *Seja Γ uma base, M um termo, $\tau \in \mathbb{T}$ e S uma substituição. Seja $UNIFY$ a função de unificação tal que $UNIFY(\tau_1, \tau_2) = S$, onde S é o unificador mais geral de τ_1 e τ_2. A função $W(\Gamma, M) = (S, \tau)$ define o algoritmo de inferência de tipos para o sistema de tipos de Damas-Milner, da seguinte forma:*

1. *Se M é uma variável x e existe uma declaração $x : \forall \alpha_1, \ldots, \alpha_n.\tau' \in \Gamma$ então S é a função identidade e $\tau = [\beta_i/\alpha_i]\tau'$, onde cada β_i é uma variável nova $(1 \leq i \leq n)$.*

2. *Se $M \equiv M_1 M_2$, seja:*

   - $W(\Gamma, M_1) = (S_1, \tau_1)$;
   - $W(S_1\Gamma, M_2) = (S_2, \tau_2)$;
   - $U = UNIFY(S_2\tau_1, \tau_2 \rightarrow \beta)$, *onde β é uma variável nova;*

   então $S = U \circ S_2 \circ S_1$ e $\tau = U\beta$;

3. *Se $M \equiv \lambda x.N$, seja β uma variável nova e $W(\Gamma_x \cup \{x : \beta\}, N) = (S_1, \tau_1)$, então $S = S_1$ e $\tau = S_1\beta \rightarrow \tau_1$;*

4. *Se $M \equiv let\ x = M_1\ in\ M_2$, seja:*

   - $W(\Gamma, M_1) = (S_1, \tau_1)$ *e*
   - $W(S_1\Gamma_x \cup \{x : \overline{S_1\Gamma}(\tau_1)\}, M_2) = (S_2, \tau_2)$

   então $S = S_2 \circ S_1$ e $\tau = \tau_2$

Exemplo 2.1.7 *Consideremos a aplicação do algoritmo acima ao termo $let\ y = (\lambda x.x)\ in\ yy$.*

$$W(\{\}, \lambda x.x) = ([\,], \alpha \rightarrow \alpha)$$

Utilizando o fecho do tipo $(\alpha \rightarrow \alpha)$, como tipo de y em $W(\{y : \forall\alpha.\alpha \rightarrow \alpha\}, yy)$ temos:

$$W(\{y : \forall\alpha.\alpha \rightarrow \alpha\}, y) = ([\,], \alpha_1 \rightarrow \alpha_1) \text{ e}$$
$$W(\{y : \forall\alpha.\alpha \rightarrow \alpha\}, y) = ([\,], \alpha_2 \rightarrow \alpha_2)$$

Dado que $U = UNIFY(\alpha_1 \rightarrow \alpha_1, (\alpha_2 \rightarrow \alpha_2) \rightarrow \beta) = [(\alpha_2 \rightarrow \alpha_2)/\alpha_1, (\alpha_2 \rightarrow \alpha_2)/\beta]$, então:

$$W(\{\}, let\ y = (\lambda x.x)\ in\ yy) = (U, (\alpha_2 \rightarrow \alpha_2))$$

Note-se que conseguimos tipificar este termo porque y é usado em (yy) com dois tipos diferentes.

Teorema 2.1.3 (Integridade) *Se $W(\Gamma, M)$ termina com (S, τ), então existe uma derivação de $S\Gamma \vdash M : \tau$.*

Definição 2.1.26 *Um esquema de tipos σ é o esquema de tipos principal de um termo M com respeito a uma base Γ, se e só se:*

- $\Gamma \vdash M : \sigma$;
- *Se $\Gamma \vdash M : \sigma'$ então σ' é uma instância genérica de σ.*

Teorema 2.1.4 *Se $W(\Gamma, M)$ termina com (S, τ), então $\overline{S\Gamma}(\tau)$ é o esquema de tipos principal de M com respeito a Γ.*

Exemplo 2.1.8 *Consideremos a aplicação do algoritmo W ao termo let $y = (\lambda x.x)$ in yy. Vimos no exemplo anterior que:*

$$W(\{\}, let\ y = (\lambda x.x)\ \ in\ yy) = ([(\alpha_2 \to \alpha_2)/\alpha_1, (\alpha_2 \to \alpha_2)/\beta], (\alpha_2 \to \alpha_2))$$

Do teorema acima resulta que:

$$\overline{S\Gamma}(\tau) = \forall \alpha_2.\alpha_2 \to \alpha_2$$

é o esquema de tipos principal de let $y = (\lambda x.x)$ in yy com respeito à base $\{\}$.

2.2 CHR *(Constraint Handling Rules)*

Nesta secção vamos apresentar o CHR *(Constraint Handling Rules)*, um mecanismo de implementação de resolutores de restrições que vai ser usado no resto da dissertação.

2.2.1 Programação por Restrições

As linguagens de programação tradicionais são limitadas nos mecanismos para especificar restrições sobre as variáveis utilizadas. Daí a necessidade de termos linguagens que permitam ao programador simplesmente declarar essas restrições, sendo da responsabilidade da implementação garantir que estas são mantidas. Dessa necessidade surgiram as linguagens de *programação por restrições*.

Vamos apresentar alguns conceitos relacionados com a programação por restrições (ver [MS98] para uma exposição mais completa do assunto).

2.2.1.1 Restrições

Como exemplo considere-se a *restrição* sobre as variáveis x e y: $x = y + 2$ onde x e $y \in \mathbb{R}$. Dada uma restrição, uma questão importante é saber se existe uma substituição das variáveis que torne a restrição verdadeira. No exemplo acima a resposta é *sim* ($x = 3$ e $y = 1$, por exemplo), mas podemos ter restrições em que a resposta é *não* (por exemplo, ($x^2 = -1$) em $\mathbb{R}$).

As restrições e o seu significado são definidos por um *domínio de restrições*. O domínio de restrições:

- Define a sintaxe das restrições.
- Determina os valores que as variáveis podem tomar, dado que, uma restrição envolvendo variáveis, pode ter significados bastante diferentes se a variável tomar por exemplo, valores inteiros, reais ou complexos.
- Define o significado de todos os símbolos, ou seja, qual o resultado de aplicar uma função aos seus argumentos e se uma dada relação é válida para determinados argumentos.

Para um dado domínio de restrições $\mathcal{D}$ as formas mais simples de restrições que podemos definir são chamadas restrições primitivas. Uma restrição primitiva é da forma $R(a_0, \ldots, a_{n-1})$, onde R é um símbolo relacional de $\mathcal{D}$ com aridade n e a_i é uma constante ou função definida em $\mathcal{D}$, ou uma variável.

Definição 2.2.1 *Uma restrição é uma fórmula de primeira ordem, construída a partir de restrições primitivas, das conectivas lógicas* $\wedge, \vee, \neg, \rightarrow, \leftrightarrow$ *e de quantificadores* $\forall, \exists$.

Definição 2.2.2 *Para um conjunto de variáveis* $V = \{x_1, \ldots, x_n\}$, *e um domínio de restrições* $\mathcal{D}$, *uma validação* $\theta = \{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$ *é uma atribuição que a cada* $x_i \in V$ *atribui um valor* $d_i \in \mathcal{D}$.

Dada uma restrição C, $\theta(C)$ *é o valor de* C *segundo a validação* θ *sobre as variáveis* V, *que se obtém substituindo cada variável pelo valor correspondente e calculando o valor da restrição obtida.*

Seja vars(C) o conjunto de variáveis que ocorrem na restrição C. Se θ *for uma validação para* $V \supseteq$ *vars(C), então* θ *é uma solução de C, se* $\theta(C)$ *for verdade no domínio de restrições em questão.*

Se uma restrição C tiver uma solução em D então é satisfazível, caso contrário é não satisfazível.

Por exemplo, dada a validação $\{p \mapsto 1000, i \mapsto 20/100\}$, e a interpretação usual dos operadores aritméticos em $\mathbb{R}$, o valor da expressão $p + i \times p$ é 1200. A validação $\{x \mapsto 7\}$ é uma solução da restrição $x \geq 4$ assim como a validação $\{x \mapsto 5, y \mapsto 7\}$, mas não a validação $\{x \mapsto 0\}$. Uma vez que tem soluções em $\mathbb{R}$ a restrição $x \geq 4$ é satisfazível. Como exemplo de uma restrição não satisfazível em $\mathbb{R}$ temos:

$$x \leq 3 \wedge x = y \wedge y \geq 4$$

2.2.1.2 Resolução de Restrições

Dada uma restrição duas questões que se colocam naturalmente são encontrar soluções para a restrição (caso existam), e descobrir se a restrição tem ou não solução. Um algoritmo que determina se uma restrição tem solução é chamado um resolutor de restrições.

Definição 2.2.3 *Um resolutor de restrições, solve, para um domínio de restrições $\mathcal{D}$ recebe como entrada uma restrição C no domínio $\mathcal{D}$ e retorna* verdadeiro *ou* falso.

Duas propriedades de um resolutor de restrições têm particular importância:

- Integridade: Se dado C solve(C) = true, então C é satisfazível.
- Completude: Se C é satisfazível então solve(C) = true.

Exemplo 2.2.1 *Consideremos o domínio constituído por termos Prolog que contém um conjunto de constantes e símbolos funcionais e o símbolo relacional binário =. Consideremos o conjunto de árvores finitas, onde cada nó de cada árvore está associado a uma constante ou a um símbolo funcional, e o número de filhos de cada nó são a aridade do símbolo funcional associado ao nó (constantes têm aridade 0 e portanto não têm filhos). Cada símbolo funcional f de aridade n é interpretado como uma árvore com n sub-árvores cuja raiz é o símbolo f e as n sub-árvores são os argumentos de f. As restrições primitivas deste domínio são equações entre termos. O domínio $\mathcal{FT}$ corresponde ao domínio de Herbrand. Exemplos de restrições neste domínio são $x = g(y)$ e $x = f(z, z) \wedge y = g(z)$.*

A resolução de restrições no domínio $\mathcal{FT}$ é a unificação [Rob65].

Exemplo 2.2.2 *Seja $\mathcal{R}$ o domínio de restrições sobre aritmética dos reais. As constantes desse domínio são os números reais, e os símbolos funcionais são as operações aritméticas usuais $+$, $-$, $\times$ e $/$. Os símbolos relacionais entre os termos aritméticos são $=$, $\geq$, $\leq$, $>$ e $<$. Uma restrição possível neste domínio é $y = 4 + x \wedge 2 \times y = x - 2$.*

Um resolutor de restrições neste domínio pode usar diferentes técnicas de resolução de acordo com as restrições a resolver. Para restrições aritméticas lineares um algoritmo usado é o Simplex, no entanto se essas restrições forem apenas equações pode ser usado o algoritmo de Gauss.

2.2.2 CHR

As linguagens de programação por restrições para determinados domínios de restrições, têm resolutores de restrições nesses domínios.

Esses resolutores não são, normalmente, manipuláveis pelo utilizador. Assim, pode tornar-se muito difícil modificar um resolutor, ou construir um resolutor para um novo domínio.

As *Constraint Handling Rules* (CHR) [Fru98] são uma extensão a linguagens existentes, que permitem escrever resolutores de restrições.

2.2.2.1 Restrições

Em CHR existem duas classes distintas de restrições: as restrições pré-definidas da linguagem, e as restrições CHR definidas pelo utilizador. No resto da tese assumimos que a linguagem usada é o Prolog. As restrições da linguagem são resolvidas por um resolutor pré-definido já existente na linguagem. As restrições CHR, são definidas pelo utilizador e são resolvidas por um programa CHR.

2.2.2.2 Sintaxe

Definição 2.2.4 *Um programa CHR é um conjunto finito de regras. Existem três tipos de regras: simplificação, propagação e simplificação/propagação. Uma regra CHR toma uma das seguintes formas:*

$$\begin{array}{lrcl}
(\mathit{Simpl}) & H_1, \ldots, H_i & \Longleftrightarrow & G_1, \ldots, G_j \mid B_1, \ldots, B_k \\
(\mathit{Prop}) & H_1, \ldots, H_i & \Longrightarrow & G_1, \ldots, G_j \mid B_1, \ldots, B_k \\
(\mathit{Simpl/Prop}) & H_1, \ldots, H_l \setminus H_{l+1}, \ldots, H_i & \Longleftrightarrow & G_1, \ldots, G_j \mid B_1, \ldots, B_k
\end{array}$$

com $i > 0, j \geq 0, k \geq 0, l > 0$, *e onde* $H_1, \ldots, H_i$ *é uma sequência não-vazia de restrições CHR,* $G_1, \ldots, G_j$ *é uma sequência, que pode ser vazia de restrições da linguagem, e* $B_1, \ldots, B_k$ *é uma sequência, que pode ser vazia, de restrições CHR e da linguagem.*

Uma sequência vazia é representada pela restrição pré-definida *true*. Se a sequência $G_1, \ldots, G_j$, for a sequência vazia (*true*), esta pode ser omitida assim como o operador $|$.

Note-se que, apesar de termos três formas para as regras, a simplificação seria suficiente, uma vez que as outras duas regras podem ser vistas como uma abreviação de uma simplificação:

$$\begin{array}{lrcl}
(\textit{Prop}) & H_1, \ldots, H_i & \Longleftrightarrow & G_1, \ldots, G_j \mid H_1, \ldots, H_i, B_1, \ldots, B_k \\
(\textit{Simpl/Prop}) & H_1, \ldots, H_l, H_{l+1}, \ldots, H_i & \Longleftrightarrow & G_1, \ldots, G_j \mid H_1, \ldots, H_l, B_1, \ldots, B_k
\end{array}$$

2.2.2.3 Semântica Declarativa

A interpretação declarativa de um programa CHR P, é dada por uma conjunção de fórmulas lógicas quantificadas universalmente (uma para cada regra), $\mathcal{P}$, e uma teoria de restrições da linguagem CT, que determina o significado das restrições da linguagem, onde ($\models$) representa a consequência lógica em CT. Fazem parte da teoria CT, obrigatoriamente uma restrição de igualdade $=$ e as restrições *true* e *false*.

Seja $\bar{x}$ a sequência de variáveis (globais) que ocorrem na sequência $H_1, \ldots, H_i$ de uma regra CHR. Sejam ainda $\bar{y}$ e $\bar{z}$ as sequência de variáveis (locais) que ocorrem, respectivamente, nas sequências $G_1, \ldots, G_j$ e $B_1, \ldots, B_k$ (assumimos, por uma questão de simplicidade, que as variáveis em $\bar{y}$ e $\bar{z}$ são disjuntas).

Definição 2.2.5 *Declarativamente, uma simplificação* CHR *é uma equivalência lógica se a sequência* $G_1, \ldots, G_j$ *for satisfeita:*

$$\forall \bar{x}\ (\exists \bar{y}\ (G_1 \wedge \cdots \wedge G_j)) \rightarrow (H_1 \wedge \cdots \wedge H_i \leftrightarrow \exists \bar{z}\ (B_1 \wedge \cdots \wedge B_k))$$

Uma propagação CHR *é uma implicação lógica se a sequência* $G_1, \ldots, G_j$ *for satisfeita:*

$$\forall \bar{x}\ (\exists \bar{y}\ (G_1 \wedge \cdots \wedge G_j)) \rightarrow (H_1 \wedge \cdots \wedge H_i \rightarrow \exists \bar{z}\ (B_1 \wedge \cdots \wedge B_k))$$

2.2.2.4 Semântica Operacional

A *semântica operacional* dos programas CHR é dada por um sistema de transições.

Definição 2.2.6 *Um estado é um tuplo anotado*

$$\langle F, E, D\rangle_{\mathcal{V}}$$

onde F é uma conjunção de restrições CHR e da linguagem, E é uma conjunção de restrições CHR, D é uma conjunção de restrições da linguagem, e a anotação $\mathcal{V}$ é uma sequência de variáveis. A leitura lógica de $\langle F, E, D\rangle_{\mathcal{V}}$ é

$$\exists \bar{y}\ F \wedge E \wedge D$$

onde $\bar{y}$ são as variáveis que ocorrem no estado, excepto as que aparecem em $\mathcal{V}$, que permanecem livres na fórmula.

Através de transições podemos passar de um estado a outro. Num estado $\langle F, E, D\rangle_{\mathcal{V}}$, F são as restrições que permanecem por resolver, e D, E são as restrições que foram acumuladas e simplificadas até ao momento.

Definição 2.2.7 *Seja P um programa* CHR *e CT a teoria de restrições para as restrições da linguagem. A relação de transição $\longmapsto$ para* CHR *é definida pelas seguintes regras:*

$$\begin{array}{rl}
\textit{Resolução} & \langle C \wedge F, E, D\rangle_{\mathcal{V}} \longmapsto \langle F, E, D'\rangle_{\mathcal{V}} \\
& \textit{se } C \textit{ é uma restrição da linguagem e } CT \models (C \wedge D) \leftrightarrow D' \\
\textit{Introdução} & \langle H \wedge F, E, D\rangle_{\mathcal{V}} \longmapsto \langle F, H \wedge E, D\rangle_{\mathcal{V}} \\
& \textit{se } H \textit{ é uma restrição CHR} \\
\textit{Simplificação} & \langle F, H' \wedge E, D\rangle_{\mathcal{V}} \longmapsto \langle B \wedge F, E, H = H' \wedge D\rangle_{\mathcal{V}} \\
& \textit{se } (H \Leftrightarrow G \mid B) \textit{ em } P \textit{ e } CT \models D \rightarrow \exists \bar{x}\ (H = H' \wedge G) \\
\textit{Propagação} & \langle F, H' \wedge E, D\rangle_{\mathcal{V}} \longmapsto \langle B \wedge F, H' \wedge E, H = H' \wedge D\rangle_{\mathcal{V}} \\
& \textit{se } (H \Rightarrow G \mid B) \textit{ em } P \textit{ e } CT \models D \rightarrow \exists \bar{x}\ (H = H' \wedge G)
\end{array}$$

Definição 2.2.8 *O estado inicial é um estado da forma*

$$\langle F, \mathrm{true}, \mathrm{true}\rangle_{\mathcal{V}}$$

onde F é uma conjunção de restrições CHR e da linguagem, e $\mathcal{V}$ é uma sequência das variáveis que ocorrem em F.

Um estado final é um estado da forma

$$\langle F, E, \mathrm{false}\rangle_{\mathcal{V}}$$

(chamado falha*) ou da forma*

$$\langle \text{true}, E, D\rangle_{\mathcal{V}}$$

em que não é possível mais passos de computação e D diferente de false *(um tal estado é chamado* sucesso*). Chamamos a F* questão*. A um estado final chama-se* resposta *à questão F.*

Tendo definido estado inicial e final podemos definir computação:

Definição 2.2.9 *A computação de uma conjunção de restrições G é uma sequência $S_0, S_1, \ldots,$ em que $S_i \mapsto S_{i+1}$, começando no estado inicial $S_0 = \langle G, \text{true}, \text{true}\rangle_{\mathcal{V}}$ e terminando num estado final. Uma computação finita é um sucesso se o estado final também o for, e falha se o estado final falhar.*

Exemplo 2.2.3 *Consideremos o seguinte programa CHR para uma restrição CHR =<.*

```
reflexividade  @ X =< Y <=> X = Y | true.
anti-simetria  @ X =< Y, Y =< X <=> X = Y.
transitividade @ X =< Y, Y =< Z ==> X =< Z.
```

A computação de $A \leq B \wedge C \leq A \wedge B \leq C$ é:

$$
\begin{array}{ll}
 & \langle A \leq B \wedge C \leq A \wedge B \leq C, \text{true}, \text{true}\rangle_{[A,B,C]} \\
\longmapsto_{\textit{Introdução}} & \langle \text{true}, A \leq B \wedge C \leq A \wedge B \leq C, \text{true}\rangle_{[A,B,C]} \\
\longmapsto_{\textit{Propagação}}\ \textbf{Transitividade} & \langle C \leq B, A \leq B \wedge C \leq A \wedge B \leq C, \text{true}\rangle_{[A,B,C]} \\
\longmapsto_{\textit{Introdução}} & \langle \text{true}, A \leq B \wedge C \leq A \wedge B \leq C \wedge C \leq B, \text{true}\rangle_{[A,B,C]} \\
\longmapsto_{\textit{Simplificação}}\ \textbf{Anti-simetria} & \langle B = C, A \leq B \wedge C \leq A, \text{true}\rangle_{[A,B,C]} \\
\longmapsto_{\textit{Resolução}} & \langle \text{true}, A \leq B \wedge C \leq A, B = C\rangle_{[A,B,C]} \\
\longmapsto_{\textit{Simplificação}}\ \textbf{Anti-simetria} & \langle A = B, \text{true}, B = C\rangle_{[A,B,C]} \\
\longmapsto_{\textit{Resolução}} & \langle \text{true}, \text{true}, A = B \wedge B = C\rangle_{[A,B,C]}
\end{array}
$$

2.2.2.5 Confluência

Num programa CHR é normal que, num dado momento da computação, seja possível aplicar mais do que uma regra a uma conjunção de restrições. No entanto, é desejável que o resultado da computação seja o mesmo, qualquer que seja a ordem pela qual as regras sejam aplicadas.

A propriedade de confluência de um programa garante que, qualquer computação a partir de um dado estado inicial, termina no mesmo estado final.

Exemplo 2.2.4 *O seguinte programa CHR,*

```
p <=> q.
p <=> false.
```

não é confluente pois, dependendo da ordem de aplicação, ***p*** *pode ser simplificado para* ***q*** *ou para* ***false****. Acrescentando uma nova regra ao programa acima, obtemos o seguinte programa confluente:*

```
p <=> q.
p <=> false.
q <=> false.
```

Em [Abd97] demonstrou-se que determinar a confluência de um programa CHR é decidível se o programa terminar.

Capítulo 3

Inferência de Tipos e Resolução de Restrições

Muitos sistemas de tipos têm algoritmos de inferência baseados na noção de resolução de restrições.

O algoritmo de inferência de tipos da linguagem ML [DM82], usa unificação para resolver restrições de igualdade entre tipos, e várias extensões deste algoritmo são baseadas em extensões de algoritmos de unificação com algum tipo de mecanismo de resolução de restrições. Como exemplo temos sistemas de tipos com estruturas [Oho95] e sistemas de tipos com sobrecarga de operadores [WB89, NP93, OWW95].

Neste contexto Odersky, Sulzmann e Wehr definiram o HM($\mathcal{X}$) [SOW99], como um esquema geral para sistemas de tipos com restrições. A instanciação do parâmetro $\mathcal{X}$ a um determinado sistema de restrições define um determinado sistema de tipos.

A inferência de tipos em HM($\mathcal{X}$) consiste na geração e resolução de um conjunto de restrições. O método de resolução de restrições depende do sistema de restrições e define o sistema de tipos.

Neste capítulo vamos descrever o HM($\mathcal{X}$), e em seguida apresentar uma implementação do algoritmo de inferência de tipos para o HM($\mathcal{X}$) usando Prolog e CHR. As restrições de igualdade entre tipos são resolvidas por unificação. Para resolver outras restrições sobre termos, definidas pelo sistema de restrições $\mathcal{X}$, é usado um resolutor de restrições escrito em CHR. A um determinado resolutor vai corresponder um determinado sistema de tipos.

3.1 HM($\mathcal{X}$)

Nesta secção vamos apresentar uma descrição do HM($\mathcal{X}$) tal como foi apresentado em [SOW99]. Os lemas e teoremas apresentados nesta secção são demonstrados em [Sul00].

3.1.1 Sistemas de Restrições

Sistemas de restrições [Sar93] constituem uma caracterização genérica algébrica do domínio e resolutores de restrições.

3.1.1.1 Sistemas de Restrições Simples

Definição 3.1.1 *Um sistema de restrições simples é uma estrutura* $(\Omega, \vdash^e)$, *onde* Ω *é um conjunto não vazio de símbolos ou restrições primitivas. A relação* $\vdash^e \subseteq p\Omega \times \Omega$ *é uma relação de derivação onde* $p\Omega$ *é o conjunto dos subconjuntos finitos de* Ω. *Chamamos a* $C \in p\Omega$ *um conjunto de restrições ou simplesmente uma restrição.*

Um sistema de restrições deve satisfazer para todas as restrições $C, D \in p\Omega$:

$$\begin{array}{ll} C \vdash^e P & \textit{se } P \in C, \\ C \vdash^e Q & \textit{se } C \vdash^e P \textit{ para todo } P \in D \textit{ e } D \vdash^e Q \end{array}$$

Vamos extender $\vdash^e$ para ser uma relação em $p\Omega \times p\Omega$ da seguinte forma: $C \vdash^e D$ se e só se $C \vdash^e P$ para todo $P \in D$. Consideremos também que $C =^e D$ se e só se $C \vdash^e D$ e $D \vdash^e C$. O termo $\vdash^e C$ é uma abreviatura de $\emptyset \vdash^e C$ e $true = \{P \mid \emptyset \vdash^e P\}$.

Exemplo 3.1.1 *Seja* Σ *o vocabulário com as constantes 0 e 1, o símbolo funcional unário* $\neg$, *os símbolos funcionais binários* $\wedge, \vee, \oplus, \Rightarrow$, *e um símbolo relacional binário* $=$. *Seja* V *um conjunto infinito de variáveis, seja* Ω *o conjunto de fórmulas do cálculo proposicional construído a partir de* $(\Sigma,\ V)$, *e* $\vdash^e$ *a validação semântica com interpretações em* $\{0, 1\}$. *O sistema* $(\Omega,\ \vdash^e)$ *é um sistema simples de restrições.*

3.1.1.2 Sistemas de Restrições Cilíndricos

Se adicionarmos a um sistema de restrições simples um operador de projecção $\exists\bar{\alpha}$, obtemos um sistema de restrições cilíndrico.

Definição 3.1.2 *Um sistema de restrições cilíndrico é uma estrutura* $\mathcal{CCS} = (\Omega, \vdash^e, Var, \{\exists\alpha \mid \alpha \in Var\})$ *tal que:*

- $(\Omega, \vdash^e)$ *é um sistema simples de restrições,*
- *Var é um conjunto infinito de variáveis,*
- *Para cada variável* $\alpha \in Var$*,* $\exists\alpha : p\Omega \to p\Omega$ *é uma operação que satisfaz:*

$$\begin{array}{lcl} C \vdash^e \exists\alpha.C & & \\ C \vdash^e D & \Rightarrow & \exists\alpha.C \vdash^e \exists\alpha.D \\ \exists\alpha.(C \wedge \exists\alpha.D) & =^e & (\exists\alpha.C) \wedge (\exists\alpha.D) \\ \exists\alpha.\exists\beta.C & =^e & \exists\beta.\exists\alpha.C \end{array}$$

Baseados no operador de projecção vamos definir o conjunto de variáveis livres $fv(C)$ e a satisfiabilidade de uma restrição C.

Definição 3.1.3 *Seja* C *uma restrição. O conjunto das variáveis que ocorrem livres em* C *é* $fv(C) = \{\alpha \mid \exists\alpha.C \neq^e C\}$*.*

Exemplo 3.1.2 *O conjunto de variáveis livres de* $C = \exists\alpha.\exists\beta.\alpha \wedge \beta \wedge \gamma$*,* $fv(C) = \{\gamma\}$*. Por exemplo, a variável* β *não ocorre livre em* C *pois:*

$$\exists\beta.C =^e \exists\beta.\exists\alpha.\exists\beta.\alpha \wedge \beta \wedge \gamma =^e \exists\alpha.\exists\beta.\exists\beta.\alpha \wedge \beta \wedge \gamma =^e \exists\alpha.\exists\beta.\alpha \wedge \beta \wedge \gamma =^e C.$$

Definição 3.1.4 *Uma restrição* C *é satisfazível se e só se* $\vdash^e \exists fv(C).C$*.*

Uma propriedade importante do operador de projecção é o facto da projecção de uma restrição não influenciar a sua satisfiabilidade.

Lema 3.1.1 *Uma restrição* C *é satisfazível se e só se* $\vdash^e \exists\alpha.C$*.*

3.1.1.3 Sistemas de Restrições sobre Termos

Um sistema de restrições cilíndrico pode ser extendido para um sistema de restrições sobre termos. Seja uma $\mathcal{T}$ álgebra de termos com assinatura $\Sigma = (Var, Cons)$, onde Var é um conjunto de variáveis e $Cons$ um conjunto de construtores de tipos contendo no mínimo o construtor $\to$ de aridade 2.

Definição 3.1.5 *Um sistema de restrições sobre termos* $\mathcal{TCS}_{\mathcal{T}} = (\Omega, \vdash^e, Var, \{\exists\alpha \mid \alpha \in Var\})$ *sobre uma álgebra de termos* $\mathcal{T}$ *é um sistema de restrições cilíndrico com predicados da forma*

$$p(\tau_1, \ldots, \tau_n) \quad (\tau_i \in \mathcal{T})$$

em que se verificam as seguintes condições:

- *Para cada par de tipos* τ, τ' *existe uma relação de igualdade* $(\tau = \tau')$ *em* $\mathcal{TCS}_{\mathcal{T}}$, *que satisfaz:*
  $\vdash^e (\alpha = \alpha)$
  $(\alpha = \beta) \vdash^e (\beta = \alpha)$
  $(\alpha = \beta) \wedge (\beta = \gamma) \vdash^e (\alpha = \gamma)$
  $(\alpha = \beta) \wedge \exists\alpha.(C \wedge (\alpha = \beta)) \vdash^e C$
  $(\tau = \tau') \vdash^e (T[\tau] = T[\tau'])$
  onde $T[]$ *é um contexto arbitrário nos termos*

- *Para cada predicado* P
  $[\tau/\alpha]P =^e \exists\alpha.(P \wedge (\alpha = \tau))$
  onde $\alpha \notin fv(\tau)$

Exemplo 3.1.3 *Para qualquer* $\mathcal{T}$ *seja Herbrand* $= (\Omega, \vdash^e, Var, \{\exists\alpha \mid \alpha \in Var\})$ *o sistema de restrições onde* Ω *contém apenas restrições primitivas da forma* $(\tau = \tau')$ *onde* τ *e* τ' *são tipos de* $\mathcal{T}$. *A igualdade em Herbrand é puramente sintática.*

Por exemplo, a equação $(f(x,y) = f(a, g(b,c))$ *dá origem a* $(x = a)$ *e* $(y = g(b,c))$. *A satisfiabilidade das restrições pode ser verificada pelo algoritmo de unificação de Robinson.*

3.1.2 Os Sistemas de Tipos HM($\mathcal{X}$)

Uma instância de HM($\mathcal{X}$) é descrita por $(\mathcal{X}, \mathcal{T}, \mathcal{S}, \Gamma_0)$. A álgebra de termos $\mathcal{T}$, descreve a linguagem de tipos e o sistema de restrições sobre termos $\mathcal{X}$, descreve a linguagem de restrições. O conjunto $\mathcal{S}$ descreve as restrições que podem aparecer nos esquemas de tipos e nas tipagens. Γ_0 representa o ambiente inicial e contém declarações de tipos para as funções primitivas da linguagem.

3.1.2.1 A linguagem de termos

A linguagem de termos é a mesma que apresentámos para o sistema de tipos de Damas-Milner.

$$M ::= x \mid \lambda x.M \mid MM' \mid \text{let } x = M \text{ in } M'$$

Construtores adicionais da linguagem são expressos como valores pré-definidos, cujos nomes e tipos são colocados no ambiente inicial Γ_0.

3.1.2.2 A linguagem de tipos

A linguagem de tipos é definida pela álgebra de termos $\mathcal{T}$, mas contém no mínimo variáveis de tipo e tipos funcionais. O conjunto de tipos é definido por:

$$\begin{aligned} \tau &::= \alpha \mid \tau \to \tau' \mid T\bar{\tau} \\ \sigma &::= \tau \mid \forall \alpha.C \Rightarrow \sigma \end{aligned}$$

onde α é uma variável de tipo, τ e τ' são tipos, e σ é um esquema de tipos. Construtores de tipos adicionais, representados na gramática por T (sendo $\bar{\tau} = \tau_1, \ldots, \tau_n$, os argumentos do construtor T de aridade n), dependem de $\mathcal{T}$, e podem variar conforme a instância HM($\mathcal{X}$).

Além disso, os esquemas de tipos $\forall \alpha.C \Rightarrow \sigma$, incluem uma restrição C, que limita os tipos que podem substituir a variável de tipo α. As restrições que aparecem nos esquemas de tipos têm que ser sempre satisfazíveis.

3.1.2.3 Regras de Inferência

Seja C uma restrição satisfazível em $\mathcal{X}$, Γ uma base, e σ um esquema de tipos. Escrevemos $C, \Gamma \vdash M : \sigma$ se obtemos $M : \sigma$ pela aplicação das seguintes regras:

$$(Var) \qquad C, \Gamma \vdash x : \sigma \quad (x : \sigma \in \Gamma)$$

$$(Abs) \qquad \frac{C, \Gamma \cup \{x : \tau\} \vdash M : \tau'}{C, \Gamma \vdash \lambda x.M : \tau \rightarrow \tau'}$$

$$(App) \qquad \frac{C, \Gamma \vdash M : \tau_1 \rightarrow \tau_2 \quad C, \Gamma \vdash M' : \tau_1}{C, \Gamma \vdash M M' : \tau_2}$$

$$(Let) \qquad \frac{C, \Gamma \vdash M : \sigma \quad C, \Gamma \cup \{x : \sigma\} \vdash M' : \tau'}{C, \Gamma \vdash \text{let } x = M \text{ in } M' : \tau'}$$

$$(\forall\ Intro) \qquad \frac{C \wedge D, \Gamma \vdash M : \tau \quad \bar{\alpha} \notin fv(C) \cup fv(\Gamma)}{C \wedge \exists \bar{\alpha}.D, \Gamma \vdash M : \forall \bar{\alpha}.D \Rightarrow \tau}$$

$$(\forall\ Elim) \qquad \frac{C, \Gamma \vdash M : \forall \bar{\alpha}.D \Rightarrow \tau' \quad C \vdash^e [\bar{\tau}/\bar{\alpha}]D}{C, \Gamma \vdash M : [\bar{\tau}/\bar{\alpha}]\tau'}$$

Uma tipagem $C, \Gamma \vdash M : \sigma$ é válida se pode ser derivada pela aplicação das regras acima e se a restrição C for satisfazível. Através da regra (∀ Intro) quantificamos algumas variáveis de tipo.

Notação: O esquema de tipos $\forall \bar{\alpha}.D \Rightarrow \tau$ é uma abreviatura de $\forall \alpha_1.D \Rightarrow \ldots \forall \alpha_n.D \Rightarrow \tau$, e $\exists \bar{\alpha}.D$ é uma abreviatura de $\exists \alpha_1. \ldots \exists \alpha_n.D$.

Podemos limitar as restrições que aparecem nos esquemas de tipos e nas derivações às chamadas restrições em *forma resolvida*. O conjunto $\mathcal{S}$, de restrições em *forma resolvida*, é sempre um subconjunto das restrições satisfazíveis em $\mathcal{X}$.

Assim, para uma restrição C, que apareça numa tipagem ou num esquema de tipos, deve existir uma restrição $D \in \mathcal{S}$, tal que $C =^e D$.

O conjunto $\mathcal{S}$ afecta o número de tipagens válidas que podemos ter para uma dada instância de HM($\mathcal{X}$). Por exemplo, se para uma dada instância HM($\mathcal{Z}$), $\mathcal{S}$ for o conjunto vazio então não é possível tipar nenhum termo em HM($\mathcal{Z}$).

O HM($\mathcal{X}$), tal como é apresentado em [SOW99], tem uma regra adicional (Sub), que permite derivar um tipo τ' para um termo M, se é possível derivar um tipo τ para M e $\tau \preceq \tau'$.

A relação $\preceq$ é determinada pelo sistema de restrições $\mathcal{X}$. A escolha da relação $\preceq$, para um dado sistema HM($\mathcal{X}$), tem que respeitar os axiomas para uma relação de ordem parcial (reflexividade, anti-simetria e transitividade) e a regra:

$$(Contr) \quad \frac{D \vdash^e (\alpha_1' \preceq \alpha_1) \quad D \vdash^e (\alpha_2 \preceq \alpha_2')}{D \vdash^e (\alpha_1 \to \alpha_2 \preceq \alpha_1' \to \alpha_2')}$$

Uma vez que nas instâncias de HM($\mathcal{X}$) que vamos estudar $\preceq$ é sempre a relação de igualdade sintáctica entre tipos (=), vamos considerar o HM($\mathcal{X}$) sem a regra (Sub).

Podemos seguramente ignorar esta regra pois, sendo $\preceq$ a igualdade sintáctica, então a regra (Sub) trivializa-se como:

$$(Sub) \quad \frac{C, \Gamma \vdash M : \tau}{C, \Gamma \vdash M : \tau}$$

Além disso, a relação de igualdade entre tipos é uma relação de ordem parcial (reflexiva, anti-simétrica e transitiva), e respeita a regra acima:

$$(Contr) \quad \frac{D \vdash^e (\alpha_1 = \alpha_1') \quad D \vdash^e (\alpha_2 = \alpha_2')}{D \vdash^e (\alpha_1 \to \alpha_2 = \alpha_1' \to \alpha_2')}$$

Exemplo 3.1.4 *O sistema de Damas-Milner é uma instância de HM($\mathcal{X}$). Seja $\mathcal{X}$ o sistema de restrições Herbrand, tal como o definimos na secção anterior. O conjunto $\mathcal{S}$ contém apenas true (que é representado pelo conjunto vazio).*

Então os únicos esquemas de tipos possíveis são da forma $\forall\alpha.\{\} \Rightarrow \sigma$, que equivalem aos esquemas de tipos $\forall\alpha.\sigma$ no sistema de Damas-Milner.

Neste caso podemos derivar $\Gamma \vdash M : \sigma$ no sistema de Damas-Milner se e só se podemos derivar $\{\}, \Gamma \vdash M : \sigma$ em HM(Herbrand).

3.1.3 Inferência de Tipos em HM($\mathcal{X}$)

A inferência de tipos nos sistemas HM($\mathcal{X}$) é feita traduzindo o problema de determinar o tipo de um termo num problema de restrições.

Assim, um termo é tipado se o problema de restrições associado tiver solução. A solução do problema de restrições é uma restrição em *forma resolvida* em $\mathcal{S}$. Se não existe uma solução para o problema de restrições então não é possível inferir um tipo para o termo.

Consideremos por exemplo, a aplicação (MN) onde M tem tipo inferido τ_1 e N tem tipo inferido τ_2. Para resolver o problema de tipificar (MN) é necessário resolver o problema de restrições $(\tau_1 = \tau_2 \to \alpha)$, onde o valor atribuído à variável nova α corresponde ao tipo do resultado da aplicação de M a N.

3.1.3.1 Normalização

A inferência de tipos em HM($\mathcal{X}$) combina geração e normalização de restrições. Vamos apresentar a normalização de restrições tal como é usada em [SOW99].

Começamos por introduzir algumas definições preliminares.

Preliminares Seja $\phi_{|U}$ a restrição da substituição ϕ ao domínio U, ou seja, $\phi_{|U}(x) = \phi(x)$ se $x \in U$ e $\phi_{|U}(x) = x$ caso contrário. Para ψ e ϕ substituições, escrevemos $\psi =_U \phi$ se e só se $\vdash^e (\psi(x) = \phi(x))$ para todo $x \in U$. Escrevemos $\psi \leq_U^{\phi'} \phi$ se e só se $\phi' \circ \psi =_U \phi$. Escrevemos $\psi \leq_U \phi$ se e só se $\exists \phi' : \psi \leq^{\phi'} \phi$. Por vezes o conjunto U é omitido.

Em HM($\mathcal{X}$) um problema de tipagem é traduzido para uma restrição C e uma substituição ψ. Chamamos ao par (C, ψ) um problema de restrições.

A normalização é a determinação de uma forma normal para um problema de restrições (C, ψ).

Definição 3.1.6 (Forma Normal) *Seja $\mathcal{X}$ um sistema de restrições, e $\mathcal{S}$ o conjunto de restrições resolvidas em $\mathcal{X}$. Seja $C \in \mathcal{S}$ e $D \in \mathcal{X}$ duas restrições e ϕ, ψ substituições. Então (C, ψ) é uma forma normal de (D, ϕ) se e só se $\phi \leq \psi, C \vdash^e \psi D$ e $\psi C =^e C$.*

Definição 3.1.7 (Forma Normal Principal) *(C, ψ) é forma normal principal de (D, ϕ) se, para todas as formas normais (C', ψ') de (D, ϕ), temos que $\psi \leq \psi'$ e $C' \vdash^e \psi' C$.*

A forma normal principal representa a melhor solução de um problema de restricões. Por exemplo, no sistema de restrições *Herbrand*, uma forma normal principal corresponde ao unificador mais geral e uma forma normal a um unificador.

Exemplo 3.1.5 *Consideremos um sistema de restrições, com restrições primitivas da forma $\alpha :: \tau$, e a seguinte regra:*

$$(\alpha :: \tau_1) \wedge (\alpha :: \tau_2) \vdash^e \tau_1 = \tau_2.$$

Então para o problema de restrições $(C, \phi) = ((\alpha :: (\beta \to \beta) \to \beta) \wedge (\alpha :: \gamma \to \beta), [])$.

- $(\alpha :: (\beta' \to \beta') \to \beta'), [\beta'/\beta, (\beta' \to \beta')/\gamma])$ *é forma normal de (C, ϕ);*
- $(\alpha :: (\beta \to \beta) \to \beta), [(\beta \to \beta)/\gamma])$ *é forma normal principal de (C, ϕ);*

A menos de renomeação de variáveis, em [Sul00] demonstrou-se que a forma normal principal de um problema de restrições é única.

Lema 3.1.2 (Unicidade) *Sejam (C, ψ) e (C', ψ'), formas normais principais de um problema de restrições (D, ϕ). Então, existe uma renomeação de variáveis ϕ' tal que $C' =^e \psi' C$ e $\psi' = \phi \circ \psi$.*

Definimos a função *normaliza* de problemas de restrições em formas normais da seguinte forma.

$$\begin{aligned} normaliza(D, \phi) &= (C, \psi) && \text{se } (C, \psi) \text{ é forma normal principal de } (D, \phi) \\ &= falha && \text{caso contrário} \end{aligned}$$

A propriedade de ter forma normal principal pode ser extendida a sistemas de restrições.

Definição 3.1.8 (Propriedade de restrição principal) *Seja $\mathcal{X}$ um sistema de restrições, e $\mathcal{S}$ o conjunto de restrições resolvidas em $\mathcal{X}$. O sistema de restrições $\mathcal{X}$ tem a propriedade de restrição principal se, para toda a restrição $D \in \mathcal{X}$ e substituição ϕ, o problema de restrições (D, ϕ), ou não tem uma forma normal ou tem uma forma normal principal.*

Por definição, um sistema de tipos HM($\mathcal{X}$) tem a propriedade de restrição principal se o sistema de restrições $\mathcal{X}$ a tiver.

3.1.3.2 Algoritmo

O algoritmo que apresentamos em seguida, é um algoritmo geral de inferência de tipos para os sistemas HM($\mathcal{X}$), apresentado em [SOW99].

Diferentes instâncias de HM($\mathcal{X}$) são resultado de diferentes sistemas de restrições.

Definição 3.1.9 *Seja ψ uma substituição, C uma restrição em forma resolvida, Γ uma base, M um termo e τ um tipo. As seguintes regras sobre $\psi, C, \Gamma \vdash^W M : \tau$ definem um algoritmo de inferência de tipos, com Γ e M como argumentos, e ψ, C e τ como resultados.*

$$(Var) \qquad \frac{x : (\forall\bar{\alpha}.D \Rightarrow \tau) \in \Gamma \qquad \bar{\beta} \; novas \qquad (C,\psi) = normaliza(D, [\bar{\beta}/\bar{\alpha}])}{\psi_{|fv(\Gamma)}, C, \Gamma \vdash^W x : \psi\tau}$$

$$(Abs) \qquad \frac{\psi, C, \Gamma_x \cup \{x : \alpha\} \vdash^W M : \tau \qquad \alpha \; nova}{\psi_{\setminus\{\alpha\}}, C, \Gamma_x \vdash^W \lambda x.M : \psi\alpha \to \tau}$$

$$(App) \qquad \frac{\begin{array}{c}\psi_1, C_1, \Gamma_x \vdash^W M_1 : \tau_1 \quad \psi_2, C_2, \Gamma_x \vdash^W M_2 : \tau_2 \\ \psi' = \psi_1 \sqcup \psi_2 \\ D = C_1 \wedge C_2 \wedge (\tau_1 = \tau_2 \to \alpha) \qquad \alpha \; nova \\ (C,\psi) = normaliza(D, \psi')\end{array}}{\psi_{|fv(\Gamma)}, C, \Gamma \vdash^W M_1 M_2 : \psi\alpha}$$

$$(Let) \qquad \frac{\begin{array}{c}\psi_1, C_1, \Gamma_x \vdash^W M_1 : \tau \qquad (C_2, \sigma) = gen(C_1, \psi_1, \tau) \\ \psi_2, C_3, \Gamma_x \cup \{x : \sigma\} \vdash^W M_2 : \tau' \\ \psi' = \psi_1 \sqcup \psi_2 \qquad D = C_2 \wedge C_3 \\ (C,\psi) = normaliza(D, \psi')\end{array}}{\psi_{|fv(\Gamma)}, C, \Gamma_x \vdash^W let \; x = M_1 \; in \; M_2 : \psi\tau'}$$

Dadas as restrições $C, C_o \in \mathcal{S}$, Γ um ambiente, e σ um esquema de tipos, a função de generalização, *gen* é definida como:

$$gen(C, \Gamma, \sigma) = (C_o, \forall\bar{\alpha}.C' \Rightarrow \sigma)$$

onde $C_o =^e \exists\bar{\alpha}.C =^e D \wedge \exists\bar{\alpha}.C'$, e $fv(D) \cap \bar{\alpha} = \emptyset$. O conjunto $\bar{\alpha}$ de variáveis é um subconjunto de $(fv(\sigma) \cup fv(C)) \setminus fv(\Gamma)$. Uma vez que C_o é uma restrição em forma resolvida, o conjunto $\bar{\alpha}$ deve garantir que $\exists\bar{\alpha}.C$ está em forma resolvida.

A função de generalização, $gen(C, \Gamma, \sigma)$, divide a restrição C em duas partes D e C'. As variáveis generalizadas aparecem apenas em C', e C' aparece como restrição no esquema de tipos resultante da generalização. Esta condição pode ser sempre satisfeita tomando D como sendo *true*. No entanto, dependendo do sistema de restrições, podem existir melhores estratégias, que mantenham a restrição que aparece no esquema de tipos generalizado, o mais pequena possível.

No algoritmo de inferência de tipos a propriedade de restrição principal dos sistemas de restrições vai estar ligada à propriedade de tipos principais dos sistemas de tipos no sentido em que, o algoritmo de inferência de tipos calcula tipos principais, se o sistema de restrições possuir a propriedade de restrição principal.

3.1.3.3 Integridade e Completude

Os resultados seguintes de integridade e completude do algoritmo de inferência $\vdash^W$ foram apresentados em [SOW99].

Teorema 3.1.1 (Integridade) *Seja M um termo e Γ uma base. Se $\psi, C, \Gamma \vdash^W M : \tau$, então $C, \psi\Gamma \vdash M : \tau, \psi C = C$, e $\psi\tau = \tau$.*

Para formular a completude do algoritmo associado ao HM($\mathcal{X}$) necessitamos das seguintes definições:

Definição 3.1.10 *Dados C uma restrição, e σ, σ' esquemas de tipos, definimos a relação $C \vdash^i \sigma \preceq \sigma'$, com o significado de σ ser um esquema de tipos mais geral que σ' com respeito à restrição C, da seguinte forma:*

$$(Sub) \qquad \frac{C \vdash^e (\tau = \tau')}{C \vdash^i \tau \preceq \tau'}$$

$$(\preceq \forall) \qquad \frac{C \wedge D \vdash^i \sigma \preceq \sigma' \qquad \alpha \notin vt(\sigma) \cup vt(C)}{C \vdash^i \sigma \preceq (\forall \alpha.D \Rightarrow \sigma')}$$

$$(\forall \preceq) \qquad \frac{C \vdash^i [\tau/\alpha]\sigma \preceq \sigma' \qquad C \vdash^e [\tau/\alpha]D}{C \vdash^i (\forall \alpha.D \Rightarrow \sigma) \preceq \sigma'}$$

onde $vt(\sigma)$ e $vt(C)$, representam as variáveis de tipo de σ e C, respectivamente.

Exemplo 3.1.6 *Consideremos os seguintes esquemas $\sigma = \forall\alpha.\{\} \Rightarrow \alpha \rightarrow \alpha$ e $\sigma' = \forall\alpha_1.\{\} \Rightarrow (\alpha_1 \rightarrow \alpha_1) \rightarrow (\alpha_1 \rightarrow \alpha_1)$. Temos:*

$$\begin{array}{c}
\vdash^e (\alpha_1 \rightarrow \alpha_1) \rightarrow (\alpha_1 \rightarrow \alpha_1) = (\alpha_1 \rightarrow \alpha_1) \rightarrow (\alpha_1 \rightarrow \alpha_1) \\
\Big\downarrow \scriptstyle (SUB) \\
\vdash^i (\alpha_1 \rightarrow \alpha_1) \rightarrow (\alpha_1 \rightarrow \alpha_1) \preceq (\alpha_1 \rightarrow \alpha_1) \rightarrow (\alpha_1 \rightarrow \alpha_1) \\
\Big\downarrow \scriptstyle (\forall \preceq) \\
\vdash^i \forall\alpha.\{\} \Rightarrow \alpha \rightarrow \alpha \preceq (\alpha_1 \rightarrow \alpha_1) \rightarrow (\alpha_1 \rightarrow \alpha_1) \\
\Big\downarrow \scriptstyle (\preceq \forall) \\
\vdash^i \forall\alpha.\{\} \Rightarrow \alpha \rightarrow \alpha \preceq \forall\alpha_1.\{\} \Rightarrow (\alpha_1 \rightarrow \alpha_1) \rightarrow (\alpha_1 \rightarrow \alpha_1)
\end{array}$$

Definição 3.1.11 *Uma base* Γ *é realizável numa restrição* C *se, para todo o par* $x : \sigma \in \Gamma$*, existe um tipo* τ *tal que* $C \vdash^i \sigma \preceq \tau$*.*

Informalmente, dizemos que o algoritmo de inferência de tipos é completo se, dada uma derivação $C', \phi\Gamma \vdash M : \sigma'$, o algoritmo de inferência tem como resultado uma restrição que é pelo menos tão pequena quanto C', e um tipo que é pelo menos tão geral quanto σ'.

Teorema 3.1.2 (Completude) *Seja* $C', \phi\Gamma \vdash M : \sigma'$ *uma tipagem, tal que* $\phi\Gamma$ *é realizável em* C'*. Então*

$$\psi, C, \Gamma \vdash^W M : \tau$$

para uma substituição ψ*, uma restrição* C*, e um tipo* τ*, tal que*

- $gen(C, \psi\Gamma, \tau) = (C_o, \sigma_o)$
- $\psi \leq^{\phi'}_{fv(\Gamma)} \phi$
- $C' \vdash^e \phi' C_0$
- $C' \vdash^i \phi' \sigma_o \preceq \sigma'$

ou seja, (C_o, σ_o, ψ) *é mais geral que* (C', σ', ϕ)*.*

Exemplo 3.1.7 *Consideremos o termo* $\lambda x.x$*. Temos:*

$$\{\}, \{\} \vdash \lambda x.x : \forall \alpha_1.\{\} \Rightarrow (\alpha_1 \to \alpha_1) \to (\alpha_1 \to \alpha_1)$$

e

$$\{\}, \{\}, \{\} \vdash^W \lambda x.x : \alpha \to \alpha$$

Temos então:

- $gen(\{\}, \{\}, \alpha \to \alpha) = (\{\}, \forall \alpha.\{\} \Rightarrow \alpha \to \alpha)$
- $\vdash^i \forall \alpha.\{\} \Rightarrow \alpha \to \alpha \preceq \forall \alpha_1.\{\} \Rightarrow (\alpha_1 \to \alpha_1) \to (\alpha_1 \to \alpha_1)$ *(ver Exemplo 3.1.6)*

As provas dos teoremas de integridade e completude do algoritmo de inferência de tipos $\vdash^W$ podem ser encontradas em [SOW99].

3.1.4 Tipos Principais

Uma propriedade desejável de um algoritmo de inferência de tipos é que retorne tipos principais. Nos sistemas de tipos HM($\mathcal{X}$), essa propriedade depende do sistema de restrições $\mathcal{X}$.

Em [Sul00] demonstrou-se que para um dado sistema de tipos HM($\mathcal{X}$), se o sistema de restrições $\mathcal{X}$ tiver a propriedade de restrição principal, então o sistema de tipos em questão tem a propriedade do tipo principal.

3.2 Inferência de Tipos e Programação por Restrições

A nossa implementação do algoritmo de inferência de tipos é feita usando Prolog e CHR. As restrições de igualdade são resolvidas usando a unificação com *occur-check* do Prolog (algoritmo de unificação de Robinson), e as restantes restrições são resolvidas por um conjunto de regras CHR. A cada sistema de tipos corresponde um conjunto diferente de regras CHR.

Comparando com a descrição do algoritmo de inferência de tipos na secção anterior temos as seguintes diferenças fundamentais:

- As substituições são feitas por propagação das ligações das variáveis, usando a unificação do Prolog;
- A normalização de restrições da forma $\tau_1 = \tau_2$ é feita usando a unificação do Prolog com *occur-check*;
- A normalização de outros tipos de restrições é feita usando CHR.

Na definição do algoritmo e a partir de agora, por abuso de notação, usamos uma sintaxe abstracta para os tipos ($\tau, \forall\bar{\alpha}.D \Rightarrow \tau, \ldots$) e termos, em vez da sintaxe correcta para termos Prolog.

Definição 3.2.1 *Dado um ambiente inicial Γ, e um termo M, o algoritmo tem como resultado o tipo T de M, e a restrição em forma resolvida C.*

$$\begin{array}{rll} \mathit{type}(C, \Gamma, x, \tau) \leftarrow & \mathit{member}(x, \Gamma, \forall\bar{\alpha}.D \Rightarrow \tau_1), & \\ & \mathit{new\_instance}(\bar{\alpha}, \tau_1, \tau, D_1, D), & (\mathit{Var}) \\ & \mathit{normalize}(D_1, C). & \end{array}$$

$$type(C, \Gamma, \lambda x.M, \alpha \to \tau) \leftarrow type(C, [(x, \forall\emptyset.\mathrm{true} \Rightarrow \alpha) \mid \Gamma], M, \tau). \quad (Abs)$$

$$\begin{aligned} type(D, \Gamma, M_1 M_2, \tau) \leftarrow\ & type(C_1, \Gamma, M_1, \tau_1), \\ & type(C_2, \Gamma, M_2, \tau_2), \\ & union(C_1, C_2, C), \qquad (Apl) \\ & unify\_with\_occurs\_check(\tau_1, \tau_2 \to \tau), \\ & normalize(C, D). \end{aligned}$$

$$\begin{aligned} type(C, \Gamma, \mathrm{let}\ x = M_1\ \mathrm{in}\ M_2, \tau) \leftarrow\ & type(C_1, \Gamma, M_1, \tau_1), \\ & gen(C_1, \Gamma, \tau_1, C_2, \sigma), \\ & type(C_3, [(x, \sigma) \mid \Gamma], M_2, \tau), \qquad (Let) \\ & union(C_2, C_3, D), \\ & normalize(D, C). \end{aligned}$$

Assumimos a representação $\forall\emptyset.true \Rightarrow \tau$ para tipos simples. Isso evita ter regras distintas para variáveis com tipos simples e esquemas de tipos.

Todas as regras fazem uso do predicado $normalize$ que apresentamos aqui:

$$\begin{aligned} normalize([\,], C) \leftarrow\ & findall\_constraints(\_, CId), \\ & removeId(CId, C). \end{aligned}$$

$$\begin{aligned} normalize([X|R1], C) \leftarrow\ & call(X), \\ & normalize(R1, C). \end{aligned}$$

A função de normalização recebe um conjunto de restrições, passa essas restrições ao CHR e no final recolhe as restrições resultantes da aplicação das regras do sistema de restrições em questão.

A função do CHR, **findall_constraints(Padrão,Lista)**, unifica Lista com uma lista de pares da forma **Restrição # Id** de restrições que encaixem com Padrão. Na função *normalize* usamos a função **findall_constraints** sem especificar Padrão para recolher todas as restrições.

Na regra (Let) do nosso algoritmo, ao tipo de M_1 é aplicada uma função de generalização, de onde se obtém um esquema de tipos para M_1 que vai ser usado na inferência de tipos para M_2.

Definimos a função generalização da seguinte forma:

$$
\begin{aligned}
gen(C, \Gamma, \tau, C_2, \forall \bar{\alpha}.C_1 \Rightarrow \tau) \leftarrow\ & fvlist(\Gamma, [\ ], L), \\
& fv(\tau, L, L_1), \\
& fvlist(C, L, L_2), \\
& union(L_1, L_2, \bar{\alpha}), \\
& split(C, \bar{\alpha}, D, C_1), \\
& newconstr(D, C_1, \bar{\alpha}, C_2).
\end{aligned}
$$

As funções *fv* e *fvlist* retornam o conjuntos de variáveis livres de um termo ou de uma lista (de restrições ou declarações de tipos), respectivamente.

A função $split(C, \bar{\alpha}, D, C_1)$ separa a restrição C em duas restrições D e C_1, tais que as variáveis em $\bar{\alpha}$ ocorrem em C_1 mas não em D. Com D, C_1 e $\bar{\alpha}$, a função $newconstr(D, C_1, \bar{\alpha}, C_2)$, constrói a restrição $C_2 = D \wedge \exists \bar{\alpha}.C_1$.

Algumas funções auxiliares do algoritmo merecem algumas considerações adicionais. Vejamos por exemplo a função *new_instance*:

$$
\begin{aligned}
new\_instance(\bar{\alpha}, \tau_1, \tau, D_1, D) \leftarrow\ & newvars(\bar{\alpha}, \bar{\beta}), \\
& replace(\tau_1, \bar{\beta}, \tau), \\
& replacelist(D_1, \bar{\beta}, D).
\end{aligned}
$$

A função $new\_instance(\bar{\alpha}, \tau_1, \tau, D_1, D)$ recebe um conjunto de variáveis $\bar{\alpha}$, um tipo τ_1 e um conjunto de restrições D_1, e retorna um tipo τ e um conjunto de restrições D, tal que $\tau = [\bar{\beta}/\bar{\alpha}]\tau_1$ e $D = [\bar{\beta}/\bar{\alpha}]D_1$ ($\bar{\beta}$ são variáveis novas).

3.3 HM($\mathcal{DM}$)

Vimos que uma instância de HM($\mathcal{X}$) é um tuplo ($\mathcal{X}$,$\mathcal{T}$,$\mathcal{S}$,Γ_0), onde $\mathcal{X}$ é o sistema de restrições, $\mathcal{T}$ a linguagem de tipos, $\mathcal{S}$ o conjunto de restrições em forma resolvida e Γ_0 o ambiente inicial.

3.3.1 O sistema de restrições $\mathcal{DM}$

Vamos considerar o sistema de restrições $\mathcal{DM}$ como sendo o sistema de restrições de *Herbrand* (ver Exemplo 3.1.3).

3.3.2 A linguagem de tipos $\mathcal{T}$

Definimos a linguagem de tipos $\mathcal{T}$ pela seguinte gramática:

$$\tau ::= \alpha \mid \tau \rightarrow \tau'$$

Ou seja, o único construtor na linguagem de tipos é o construtor $\rightarrow$ de aridade 2.

3.3.3 O conjunto de restrições em forma resolvida $\mathcal{S}$

O conjunto $\mathcal{S}$ de restrições em forma resolvida consiste em todas as restrições satisfazíveis da forma:

$$C ::= \mathit{true}$$

onde *true* é representado pelo conjunto vazio. Ou seja, os únicos esquemas de tipos possíveis são da forma $\forall\alpha.\{\} \Rightarrow \sigma$, que correspondem aos esquemas de tipos $\forall\alpha.\sigma$ do sistema de Damas-Milner.

3.3.4 O ambiente inicial Γ_0

Vimos que o ambiente inicial contém as declarações de tipo das funções primitivas da linguagem em questão, assim como de construtores adicionais.

Não vamos considerar construtores adicionais para o HM($\mathcal{DM}$). Ou seja, a linguagem de termos é exactamente a mesma que definimos para o HM($\mathcal{X}$) geral.

Imaginemos que queríamos extender a linguagem para lidar com funções de comparação $(eq, <, >, \leq, \geq)$. Mantendo a linguagem de termos igual, consideraríamos essas funções como sendo variáveis declaradas no ambiente inicial. Teríamos então o ambiente inicial $\Gamma_0 = [(eq, \forall\alpha.\text{true} \Rightarrow \alpha \rightarrow \alpha \rightarrow \text{bool}), \ldots, (\geq, \forall\alpha.\text{true} \Rightarrow \alpha \rightarrow \alpha \rightarrow \text{bool})]$.

3.3.5 Exemplos

Exemplo 3.3.1 *Consideremos o seguinte programa:*

$$
\begin{aligned}
f \;=\; & \textbf{let} \\
& \quad g\ x \;=\; x \\
& \textbf{in} \\
& g\ g
\end{aligned}
$$

Aplicando o algoritmo a $g = (\lambda x.x)$ *temos:*

$type(C, \Gamma_0, (\lambda x.x), \alpha \rightarrow \tau)$

$\quad type(C, [(x, \alpha) \mid \Gamma_0], x, \tau)$

$\quad\quad member(x, [(x, \alpha) \mid \Gamma_0], \beta)$

$\quad\quad new\_instance(\{\}, \alpha, \tau, \{\}, D)$ $\qquad \tau = \alpha, D = \{\}$

$\quad\quad \boldsymbol{normalize}(\{\}, C)$ $\qquad C = \{\}$

ou seja,

$type(C, \Gamma_0, (\lambda x.x), \alpha \rightarrow \alpha)$

Consideremos agora a aplicação do algoritmo a $let\ g = (\lambda x.x)\ in\ gg$

$type(C, \Gamma_0, let\ g = (\lambda x.x)\ in\ gg, \tau)$

$\quad type(C_1, \Gamma_0, (\lambda x.x), \tau_1)$ $\qquad \tau_1 = \alpha \rightarrow \alpha, C_1 = \{\}$

$\quad gen(C_1, \Gamma_0, \alpha \rightarrow \alpha, C_2, \sigma)$ $\qquad C_2 = \{\}, \sigma = \forall \alpha.\{\} \Rightarrow \alpha \rightarrow \alpha$

$\quad type(C, [(g, \forall \alpha.\{\} \Rightarrow \alpha \rightarrow \alpha) \mid \Gamma_0], gg, \tau)$

$\quad\quad type(C_3, [(g, \forall \alpha.\{\} \Rightarrow \alpha \rightarrow \alpha) \mid \Gamma_0], g, \tau_3)$

$\quad\quad\quad member(g, [(g, \forall \alpha.\{\} \Rightarrow \alpha \rightarrow \alpha) \mid \Gamma_0], \forall \alpha.\{\} \Rightarrow \alpha \rightarrow \alpha)$

$\quad\quad\quad new\_instance(\alpha, \alpha \rightarrow \alpha, \tau_3, \{\}, D_3)$ $\qquad \tau_3 = \alpha_1 \rightarrow \alpha_1, D_3 = \{\}$

$\quad\quad\quad \boldsymbol{normalize}(\{\}, C_3)$ $\qquad C_3 = \{\}$

$\quad\quad type(C_4, [(g, \forall \alpha.\{\} \Rightarrow \alpha \rightarrow \alpha) \mid \Gamma_0], g, \tau_4)$

$\quad\quad\quad member(g, [(g, \forall \alpha.\{\} \Rightarrow \alpha \rightarrow \alpha) \mid \Gamma_0], \forall \alpha.\{\} \Rightarrow \alpha \rightarrow \alpha)$

$\quad\quad\quad new\_instance(\alpha, \alpha \rightarrow \alpha, \tau_4, \{\}, D_4)$ $\qquad \tau_4 = \alpha_2 \rightarrow \alpha_2, D_4 = \{\}$

$\quad\quad\quad \boldsymbol{normalize}(\{\}, C_4)$ $\qquad C_4 = \{\}$

$\quad\quad union(\{\}, \{\}, D)$ $\qquad D = \{\}$

$\quad\quad unify\_with\_occurs\_check(\alpha_1 \rightarrow \alpha_1, (\alpha_2 \rightarrow \alpha_2) \rightarrow \alpha_3)$ $\qquad \alpha_1 = \alpha_2 \rightarrow \alpha_2, \alpha_3 = \alpha_2 \rightarrow \alpha_2$

$\quad\quad \boldsymbol{normalize}(\{\}, C)$ $\qquad C = \{\}$

$\quad\quad \tau = \alpha_3$ $\qquad \tau = \alpha_2 \rightarrow \alpha_2$

Ou seja,

$type(\{\}, \Gamma_0, let\ g = (\lambda x.x)\ \ in\ gg, \alpha_2 \to \alpha_2)$

Os tipos inferidos para f *e* g *com ambiente inicial* $\Gamma_0 = \emptyset$, *são respectivamente:*

$$\begin{array}{l} f\ ::\ \forall\, \alpha_2\, .\{\}\ \Rightarrow\ \alpha_2\ \to\ \alpha_2 \\ g\ ::\ \forall\, \alpha\, .\{\}\ \Rightarrow\ \alpha\ \to\ \alpha \end{array}$$

Exemplo 3.3.2 *Consideremos o seguinte programa:*

$$\begin{array}{lcl} h\ x\ y & = & \mathbf{let} \\ & & \quad g\ x\ y\ =\ x \\ & & \mathbf{in} \\ & & g\ (f\ x)\ (f\ y) \end{array}$$

Os tipos inferidos para h *e* g, *com o ambiente inicial* $\Gamma_0 = \{f : \forall\alpha.\{\} \Rightarrow \alpha \to \alpha\}$, *são:*

$$\begin{array}{l} g\ ::\ \forall\, \alpha,\, \beta.\{\}\ \Rightarrow\ \alpha\ \to\ \beta\ \to\ \alpha \\ h\ ::\ \forall\, \alpha,\, \beta.\{\}\ \Rightarrow\ \alpha\ \to\ \beta\ \to\ \alpha \end{array}$$

E, com o ambiente inicial $\Gamma_0 = \{f : \forall\alpha.\{\} \Rightarrow \alpha \to \gamma\}$, *são:*

$$\begin{array}{l} g\ ::\ \forall\, \alpha,\, \beta.\{\}\ \Rightarrow\ \alpha\ \to\ \beta\ \to\ \alpha \\ h\ ::\ \forall\, \alpha,\, \beta.\{\}\ \Rightarrow\ \alpha\ \to\ \beta\ \to\ \gamma \end{array}$$

Capítulo 4

Estruturas com Campos

Neste capítulo vamos descrever o sistema de tipos de Ohori [Oho95], que extende o sistema de tipos da linguagem ML, para lidar com estruturas.

Em seguida, apresentamos o sistema de Ohori como uma instância do HM($\mathcal{X}$), definindo as regras de normalização de restrições para este sistema, com base num sistema de restrições apresentado em [Sul00].

4.1 O sistema $\mathcal{O}$ de Ohori

Em [Oho95] foi descrito o sistema de tipos polimórfico que vamos descrever em seguida, para linguagens com estruturas.

4.1.1 Linguagem de termos

A linguagem de termos é uma extensão à linguagem de termos do sistema de Damas-Milner com termos que tratam da criação, selecção e alteração de estruturas. Assim, a linguagem de termos tem a seguinte sintaxe:

$$\begin{array}{rcl} M & ::= & x \mid MM \mid \lambda x.M \mid let\ x = M\ in\ M \\ & \mid & \{l = M, \ldots, l = M\} \mid M.l \mid \mathit{modify}(M, l, M) \end{array}$$

O termo $\{l_1 = M_1, \ldots, l_n = M_n\}$ representa a criação de uma estrutura com n campos $l_1, \ldots, l_n$ sendo o valor de cada campo $M_1, \ldots, M_n$ respectivamente.

O termo $M.l$ representa a selecção do campo l da estrutura M, e o termo $\mathit{modify}(M_1, l, M_2)$ representa a alteração do valor do campo l da estrutura M_1 para M_2.

Exemplo 4.1.1 *Como exemplo de um termo da linguagem descrita acima consideremos o seguinte:*

$$
\begin{array}{l}
\lambda\, x\; y\, .\, \mathbf{let} \\
\qquad\qquad \mathit{nome} \;=\; \lambda\, z.\, z.\mathit{Nome} \\
\qquad\quad \mathbf{in} \\
\qquad\quad \mathit{nome}\, \{\mathit{Nome} \;=\; x,\; \mathit{Idade} \;=\; y\}
\end{array}
$$

4.1.2 Tipos e Espécies

O conjunto de tipos deste sistema é dado pela seguinte gramática:

$$
\begin{array}{rcl}
\tau & ::= & \alpha \mid \tau \to \tau' \mid \{l : \tau_1, \ldots, l : \tau_n\} \\
\sigma & ::= & \tau \mid \forall \alpha :: k.\sigma
\end{array}
$$

onde τ representa tipos monomórficos e σ representa tipos polimórficos.

O tipo $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ representa as estruturas que contêm os campos $l_1, \ldots, l_n$ com tipos $\tau_1, \ldots, \tau_n$ respectivamente.

Exemplo 4.1.2 *O seguinte termo de criação de uma estrutura*

$$\{\mathit{Nome} = x, \mathit{Idade} = y\}$$

tem tipo

$$\{\mathit{Nome} : \alpha, \mathit{Idade} : \beta\}$$

onde α e β são os tipos de x e y respectivamente.

As variáveis de tipo nos esquemas de tipos aparecem restringidas a um conjunto de tipos denominado *espécie*. O conjunto de *espécies* é dado pela seguinte gramática:

$$k ::= U \mid \{\!\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\!\}$$

U representa o conjunto de todos os tipos. A espécie $\{\!\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\!\}$ representa as estruturas que contêm, no mínimo, os campos $l_1, \ldots, l_n$ com tipos $\tau_1, \ldots, \tau_n$ respectivamente.

Exemplo 4.1.3 *Se uma variável de tipo α tem espécie U, então α pode representar qualquer tipo.*

Se α tiver espécie

$$\{\!\{Nome : string, Idade : int\}\!\}$$

então α pode representar o tipo

$$\{Nome : string, Idade : int, Telefone : int\}$$
$$ou$$
$$\{Nome : string, Idade : int, Morada : string\}$$

mas não,

$$\{Nome : string, Telefone : int, Morada : string\}$$

Como as variáveis de tipo nos esquemas de tipos aparecem restringidas a uma espécie, é necessário distinguir quais as variáveis que aparecem livres e quais as que aparecem restringidas.

Definição 4.1.1 *O conjunto das variáveis de tipo livres de um tipo σ, $FTV(\sigma)$ é definido da seguinte forma:*

$$\begin{array}{rcl} FTV(\alpha) & = & \{\alpha\} \\ FTV(\tau_1 \to \tau_2) & = & FTV(\tau_1) \cup FTV(\tau_2) \\ FTV(\{l_1 : \tau_1, \ldots, l_n : \tau_n\}) & = & FTV(\tau_1) \cup \cdots \cup FTV(\tau_n) \\ \\ FTV(\forall\alpha :: \kappa.\sigma) & = & FTV(\kappa) \cup (FTV(\sigma) \setminus \{\alpha\}) \end{array}$$

O conjunto de variáveis de tipo livres de uma espécie κ, $FTV(\kappa)$ é definido da seguinte forma:

$$FTV(\{\!\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\!\}) \quad = \quad FTV(\tau_1) \cup \cdots \cup FTV(\tau_n)$$

Exemplo 4.1.4 *O conjunto de variáveis de tipo livres de $\sigma = \forall\alpha :: \{\!\{l : \alpha_1 \to \alpha_2\}\!\}.\alpha \to \beta$, é $FTV(\sigma) = \{\alpha_1, \alpha_2, \beta\}$*

Definição 4.1.2 *Seja α uma variável de tipo e κ uma espécie. Temos que:*

- *Um ambiente $\mathcal{K}$ é um conjunto de declarações da forma $\alpha :: \kappa$.*
- *O domínio de um ambiente $\mathcal{K} = \{\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n\}$, $dom(\mathcal{K})$, é $\{\alpha_1, \ldots, \alpha_n\}$, e $\kappa_i = \mathcal{K}(\alpha_i)$.*

Neste sistema de tipos as variáveis de tipo livres devem aparecer restringidas por uma espécie num ambiente $\mathcal{K}$.

Definição 4.1.3 *Um tipo σ é bem formado com respeito a um ambiente $\mathcal{K}$, se $FTV(\sigma) \subseteq dom(\mathcal{K})$.*

Esta noção extende-se de forma natural a espécies.

Definição 4.1.4 *Uma espécie κ é bem formada com respeito a um ambiente $\mathcal{K}$, se $FTV(\kappa) \subseteq dom(\mathcal{K})$.*

As variáveis de tipo que aparecem livres em espécies em $\mathcal{K}$ também devem aparecer restringidas em $\mathcal{K}$. Isso corresponde à noção de ambiente bem formado, e é formalizado pela seguinte condição:

Definição 4.1.5 *Um ambiente $\mathcal{K}$ é bem formado se, para todo o $\alpha \in dom(\mathcal{K})$, $FTV(\mathcal{K}(\alpha)) \subseteq dom(\mathcal{K})$.*

Definição 4.1.6 *$\mathcal{K}\{\alpha :: \kappa\}$ representa $\mathcal{K} \cup \{\alpha :: \kappa\}$, se $\mathcal{K}$ é bem formado, $\alpha \notin dom(\mathcal{K})$, e $FTV(\kappa) \subseteq dom(\mathcal{K})$.*

A relação $\mathcal{K} \vdash \tau :: \kappa$ representa o facto de τ ter espécie κ com respeito a $\mathcal{K}$.

Definição 4.1.7 *Seja τ um tipo, κ uma espécie, e $\mathcal{K}$ um ambiente. Dizemos que τ tem espécie κ com respeito ao ambiente $\mathcal{K}$, $\mathcal{K} \vdash \tau :: \kappa$, se $\tau :: \kappa$ pode ser obtido pela aplicação das seguintes regras:*

$$\mathcal{K} \vdash \tau :: U \text{ para todo o } \tau \text{ bem formado com respeito a } \mathcal{K}$$

$$\mathcal{K} \vdash t :: \{\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\} \text{ se } \mathcal{K}(t) = \{\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\}$$

$$\begin{array}{l} \mathcal{K} \vdash \{l_1 : \tau_1, \ldots, l_n : \tau_n, \cdots\} :: \{\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\} \\ \qquad \text{se } \{l_1 : \tau_1, \ldots, l_n : \tau_n, \cdots\} \text{ é bem formado com respeito a } \mathcal{K} \end{array}$$

Note-se que, se $\mathcal{K} \vdash \sigma :: \kappa$ então, σ e κ são bem formados com respeito a $\mathcal{K}$.

Exemplo 4.1.5 *Pelas regras acima definidas temos:*

$$\{\alpha_1 :: \{\!\{Idade : \alpha_2\}\!\}, \alpha_2 :: U\} \vdash \alpha_1 \rightarrow \alpha_2 :: U$$

e para $\mathcal{K} = \{\alpha_2 :: U\}$

$$\mathcal{K} \nvdash \{Nome : \alpha_1, Idade : \alpha_2\} :: \{\!\{Nome : \alpha_1, Idade : \alpha_2\}\!\}$$

pois $\{Nome : \alpha_1, Idade : \alpha_2\}$ *não é bem formado com respeito a* $\mathcal{K}$*, uma vez que* $FTV(\{Nome : \alpha_1, Idade : \alpha_2\}) = \{\alpha_1, \alpha_2\} \nsubseteq \{\alpha_2\} = dom(\mathcal{K})$.

Também a noção de substituição deve ter em consideração as restrições de espécie.

Definição 4.1.8 *Uma substituição* S*, é bem formada com respeito a um ambiente* $\mathcal{K}$ *se, para qualquer* $\alpha \in dom(S)$, $S(\alpha)$ *é bem formado com respeito a* $\mathcal{K}$.

Definição 4.1.9 *Uma substituição restringida a um ambiente é um par* $(\mathcal{K}, S)$*, sendo* $\mathcal{K}$ *um ambiente e* S *uma substituição bem formada com respeito a* $\mathcal{K}$*. O ambiente* $\mathcal{K}$ *em* $(\mathcal{K}, S)$ *especifica restrições de espécie do resultado da substituição.*

Uma substituição $(\mathcal{K}_1, S)$ *respeita um ambiente* $\mathcal{K}_2$ *se, para qualquer* $\alpha \in dom(\mathcal{K}_2)$, $\mathcal{K}_1 \vdash S(\alpha) :: S(\mathcal{K}_2(\alpha))$*. Ou seja, se* $(\mathcal{K}_1, S)$ *respeita* $\mathcal{K}_2$*, então a substituição pode ser aplicada a um tipo* σ *restringido por* $\mathcal{K}_2$*, sendo o resultado um tipo* $S(\sigma)$ *restringido por* $\mathcal{K}_1$.

Exemplo 4.1.6 *Seja* $\mathcal{K} = \{\beta :: U, \gamma :: U\}$ *e* $S = [\{l_1 : \beta, l_2 : \gamma\}/\alpha]$*. É verdade que*

$$(\mathcal{K}, S) \text{ respeita } \mathcal{K}' = \{\alpha :: \{\!\{l_1 : \beta, l_2 : \gamma\}\!\}, \beta :: U, \gamma :: U\}$$

pois para $dom(\mathcal{K}') = \{\alpha, \beta, \gamma\}$*, temos:*

$$\mathcal{K} \vdash \{l_1 : \beta, l_2 : \gamma\} :: \{\!\{l_1 : \beta, l_2 : \gamma\}\!\}$$
$$\mathcal{K} \vdash \beta :: U$$
$$\mathcal{K} \vdash \gamma :: U$$

Neste sistema os tipos dependem de variáveis de tipo para além das suas próprias variáveis livres. Assim torna-se necessário extender a noção de variável livre de um tipo.

Definição 4.1.10 *Para um tipo σ bem formado com respeito a $\mathcal{K}$, o conjunto de variáveis de tipo essencialmente livres de σ com respeito a $\mathcal{K}$, denotado por $EFTV(\mathcal{K}, \sigma)$, é o menor conjunto tal que:*

- $FTV(\sigma) \subseteq EFTV(\mathcal{K}, \sigma)$
- *se* $\alpha \in EFTV(\mathcal{K}, \sigma)$ *então* $FTV(\mathcal{K}(\alpha)) \subseteq EFTV(\mathcal{K}, \sigma)$

Ou seja, $\alpha \in EFTV(\mathcal{K}, \sigma)$ *se* α *é livre em* σ *ou em restrições em* $\mathcal{K}$.

Exemplo 4.1.7 *A variável de tipo* α_1 *é essencialmente livre em* α_2 *com respeito a* $\mathcal{K} = \{\alpha_1 :: U, \alpha_2 :: \{\{l : \alpha_1\}\}\}$

4.1.3 Regras de inferência

Se recordarmos as regras de inferência para o sistema de Damas-Milner podemos observar a existência de uma regra de instanciação. Neste sistema a instanciação é dada pela regra para as variáveis, através da condição $\Gamma(x) \geq \tau$, cujo significado é "τ é instância genérica de $\Gamma(x)$".

Aqui a noção de instância genérica tem em consideração o ambiente $\mathcal{K}$.

Definição 4.1.11 *Seja* σ_1 *um tipo bem formado com respeito a* $\mathcal{K}$. *Dizemos que* σ_2 *é uma instância genérica de* σ_1 *com respeito a* $\mathcal{K}$, *notação* $\mathcal{K} \vdash \sigma_1 \geq \sigma_2$, *se* $\sigma_1 = \forall\alpha_1 :: \kappa_1^1 \cdots \forall\alpha_n :: \kappa_n^1.\tau_1$, $\sigma_2 = \forall\beta_1 :: \kappa_1^2 \cdots \forall\beta_m :: \kappa_m^2.\tau_2$, *e existe uma substituição* S *tal que* $dom(S) = \{\alpha_1^1, \ldots, \alpha_n^1\}$, $(\mathcal{K}\{\beta_1 :: \kappa_1^2, \ldots, \beta_m :: \kappa_m^2\}, S)$ *respeita* $\mathcal{K}\{\alpha_1 :: \kappa_1^1, \ldots, \alpha_n :: \kappa_n^1\}$, *e* $\tau_2 = S(\tau_1)$.

Exemplo 4.1.8 *O tipo* $\alpha \to \{l_1 : \alpha, l_2 : \beta\}$ *é uma instância genérica de* $\forall\alpha_1 :: U.\forall\alpha_2 :: \{\{l_1 : \alpha, l_2 : \beta\}\}.\alpha_1 \to \alpha_2$, *com respeito a* $\mathcal{K} = \{\alpha : U, \beta : U\}$. *Seja* $S = [\alpha/\alpha_1, \{l_1 : \alpha, l_2 : \beta\}/\alpha_2]$, *temos que:*

- $(\mathcal{K}, S)$ *respeita* $\mathcal{K}\{\alpha_1 :: U, \alpha_2 :: \{\{l_1 : \alpha, l_2 : \beta\}\}\}$;
- $S(\alpha_1 \to \alpha_2) = \alpha \to \{l_1 : \alpha, l_2 : \beta\}$.

Outra condição imposta na regra para as variáveis é que a base Γ seja bem formada com respeito ao ambiente $\mathcal{K}$, ou seja, para qualquer $x \in dom(\Gamma)$, $\Gamma(x)$ é bem formado com respeito a $\mathcal{K}$. Segundo a definição de instância genérica, em $\mathcal{K} \vdash \sigma \geq \tau$, σ tem que ser bem formado com respeito a $\mathcal{K}$.

Na definição do sistema de tipos vai ser necessário redefinir a noção de fecho de modo a considerar o ambiente $\mathcal{K}$.

Definição 4.1.12 *Sejam Γ e τ bem formados com respeito a $\mathcal{K}$. O fecho de τ com respeito a Γ e $\mathcal{K}$, notação $Cls(\mathcal{K}, \Gamma, \tau)$, é um par $(\mathcal{K}', \forall\alpha_1 :: \kappa_1 \cdots \forall\alpha_n :: \kappa_n.\tau)$ tal que $\mathcal{K}'\{\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n\} = \mathcal{K}$, e $\{\alpha_1, \ldots, \alpha_n\} = EFTV(\mathcal{K}, \tau) \setminus EFTV(\mathcal{K}, \Gamma)$.*

Exemplo 4.1.9 *O fecho de $(\alpha_1 \to \alpha_2)$ com respeito a $\Gamma = \{x : \alpha \to \beta, y : \beta\}$, $\mathcal{K} = \{\alpha_1 :: U, \alpha_2 :: \{\!\{l_1 : \alpha, l_2 : \beta\}\!\}, \alpha :: U, \beta :: U\}$ é:*

$$Cls(\mathcal{K}, \Gamma, \alpha_1 \to \alpha_2) = (\{\alpha :: U, \beta :: U\},\ \forall\alpha_1 :: U.\forall\alpha_2 :: \{\!\{l_1 : \alpha, l_2 : \beta\}\!\}.\alpha_1 \to \alpha_2).$$

Seja $\mathcal{K}$ um ambiente, Γ uma base, e σ um esquema de tipos. Escrevemos $\mathcal{K}, \Gamma \vdash M : \sigma$ se obtemos $M : \sigma$ pela aplicação das seguintes regras:

$$(Ax) \qquad \mathcal{K}, \Gamma \vdash x : \tau, \text{ se } \mathcal{K} \vdash \Gamma(x) \geq \tau$$

$$(Gen) \qquad \frac{\mathcal{K}, \Gamma \vdash M : \tau \quad Cls(\mathcal{K}, \Gamma, \tau) = (\mathcal{K}', \sigma)}{\mathcal{K}', \Gamma \vdash M : \sigma}$$

$$(App) \qquad \frac{\mathcal{K}, \Gamma \vdash M : (\tau' \to \tau) \quad \mathcal{K}, \Gamma \vdash M' : \tau'}{\mathcal{K}, \Gamma \vdash (MM') : \tau}$$

$$(Abs) \qquad \frac{\mathcal{K}, \Gamma \cup \{x : \tau'\} \vdash M : \tau}{\mathcal{K}, \Gamma \vdash \lambda x.M : (\tau' \to \tau)}$$

$$(Let) \qquad \frac{\mathcal{K}, \Gamma \vdash M : \sigma \quad \mathcal{K}, \Gamma \cup \{x : \sigma\} \vdash M' : \tau}{\mathcal{K}, \Gamma \vdash let\ x = M\ in\ M' : \tau}$$

$$(Rec) \qquad \frac{\mathcal{K}, \Gamma \vdash M_i : \tau_i (1 \leq i \leq n)}{\mathcal{K}, \Gamma \vdash \{l_1 = M_1, \ldots, l_n = M_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}$$

$$(Selec) \qquad \frac{\mathcal{K}, \Gamma \vdash M : \tau \quad \mathcal{K} \vdash \tau :: \{\!\{l : \tau'\}\!\}}{\mathcal{K}, \Gamma \vdash M.l : \tau'}$$

$$(Modif) \qquad \frac{\mathcal{K}, \Gamma \vdash M : \tau \quad \mathcal{K}, \Gamma \vdash M' : \tau' \quad \mathcal{K} \vdash \tau :: \{\!\{l : \tau'\}\!\}}{\mathcal{K}, \Gamma \vdash modify(M, l, M') : \tau'}$$

Exemplo 4.1.10 *Para o termo* $\{N = x, I = y\}.N$*, onde* x, y *têm tipos* α, β *respectivamente, e* $\mathcal{K} = \{\alpha :: U, \beta :: U\}$ *temos:*

$$
\begin{tikzcd}
\mathcal{K}, \{x : \alpha, y : \beta\} \vdash x : \alpha \arrow[dr, "REC"'] & & \mathcal{K}, \{x : \alpha, y : \beta\} \vdash y : \beta \arrow[dl, "REC"] & \\
& \mathcal{K}, \{x : \alpha, y : \beta\} \vdash \{N = x, I = y\} : \{N : \alpha, I : \beta\} \arrow[dr, "SEL"'] & & \mathcal{K} \vdash \{N : \alpha, I : \beta\} :: \{\!\{N :: \alpha\}\!\} \arrow[dl, "SEL"] \\
& & \mathcal{K}, \{x : \alpha, y : \beta\} \vdash \{N = x, I = y\}.N : \alpha &
\end{tikzcd}
$$

4.2 O HM($\mathcal{R}$)

Nesta secção, apresentamos o sistema $\mathcal{O}$ de Ohori como uma instância do HM($\mathcal{X}$), definindo as regras CHR usadas na normalização das restrições nos tipos.

A descrição do sistema de restrições em questão foi apresentada em [Sul00], para ilustrar o uso de sistemas de restrições específicos para inferência de tipos. Nesse trabalho foi demonstrada a equivalência entre a instância do HM($\mathcal{X}$) e o sistema de Ohori no sentido em que, todo o programa tipável no sistema de Ohori é tipável em HM($\mathcal{R}$) e vice-versa.

4.2.1 A linguagem de tipos $\mathcal{T}$

Como vimos na secção anterior, o sistema de Ohori tem, para além de variáveis de tipo e funções, estruturas com campos. Assim, definimos a linguagem de tipos $\mathcal{T}$ pela seguinte gramática:

$$\tau ::= \alpha \mid \tau \to \tau' \mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$$

onde, como já vimos, o tipo $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ representa as estruturas com campos $l_1, \ldots, l_n$, cujos tipos são respectivamente $\tau_1, \ldots, \tau_n$.

Tal como no sistema de Ohori, as variáveis de tipo nos esquemas de tipos $\forall \bar{\alpha}.D \Rightarrow \tau$ podem aparecer restringidas a uma *espécie* κ. Em HM($\mathcal{R}$) uma *espécie* κ é da forma $\langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle$, e representa todas as estruturas que contêm, no mínimo, os campos $l_1, \ldots, l_n$.

4.2.2 A linguagem de termos e o ambiente inicial Γ_0

Quando apresentámos o HM($\mathcal{X}$), dissemos que construtores adicionais e funções primitivas da linguagem são tratados como variáveis, cuja declaração de tipos aparece no ambiente inicial Γ_0.

Também vimos que o sistema de Ohori extende a linguagem de termos para lidar com criação de estruturas e selecção e alteração de campos de estruturas.

Assim os termos de criação de estruturas e de selecção e alteração de campos, vão ser tratados pelo algoritmo de inferência de tipos como variáveis cujas declarações adicionadas ao ambiente inicial passamos a explicar.

Seja $l_1, \ldots, l_n$ uma sequência de campos, e $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ o tipo que representa uma estrutura com esses campos. Consideramos para tal sequência o construtor $l_1\_\ldots\_l_n$ e adicionamos a Γ_0:

$$l_1\_\ldots\_l_n : \tau_1 \to \cdots \to \tau_n \to \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$$

Assim, $l_1\_\ldots\_l_n e_1 \ldots e_n$ representa a criação da estrutura $\{l_1 = e_1, \ldots, l_n = e_n\}$.

Exemplo 4.2.1 *Para a estrutura com campos Nome e Idade vamos ter a seguinte declaração:*

$$Nome\_Idade : \tau_1 \to \tau_2 \to \{Nome : \tau_1, Idade : \tau_2\}$$

A aplicação $(Nome\_Idade\ x\ y)$ *em que* x *tem tipo* γ *e* y *tem tipo* δ, *tem como resultado a criação da estrutura:*

$$\{Nome = x, Idade = y\}$$

com tipo $\{Nome : \gamma, Idade : \delta\}$.

Para cada campo l adicionamos ao ambiente inicial Γ_0 a função:

$$\_l : \forall \alpha, \beta.(\alpha :: \langle l : \beta \rangle) \Rightarrow \alpha \to \beta$$

que corresponde à selecção do campo l. A função $\_l$ recebe como argumento α cuja *espécie* é $\langle l : \beta \rangle$, ou seja, uma estrutura que tem no mínimo o campo l com tipo β. O resultado da função $\_l$ é do tipo β, o que seria de esperar uma vez que o campo l tem tipo β.

Exemplo 4.2.2 *Se considerarmos a estrutura do exemplo anterior vamos ter as seguintes declarações no ambiente inicial:*

$$\_.Nome : \forall \alpha, \beta.(\alpha :: \langle Nome : \beta \rangle) \Rightarrow \alpha \to \beta$$
$$e$$
$$\_.Idade : \forall \alpha, \beta.(\alpha :: \langle Idade : \beta \rangle) \Rightarrow \alpha \to \beta$$

A aplicação $(\_.Idade\ (Nome\_Idade\ x\ y))$ *em que* x *tem tipo* γ *e* y *tem tipo* δ*, tem como resultado* y *de tipo* δ*.*

Para cada campo l adicionamos ainda a Γ_0 a função de alteração do campo l.

$$altera_l : \forall \alpha, \beta.(\alpha :: \langle l : \beta \rangle) \Rightarrow \alpha \to \beta \to \alpha$$

A função tem como argumentos uma estrutura com um campo l com tipo β, e um segundo argumento do tipo β. Como resultado a função de modificação tem uma estrutura do mesmo tipo do primeiro argumento.

Exemplo 4.2.3 *Para a mesma estrutura dos exemplos anteriores vamos ter as seguintes declarações no ambiente inicial:*

$$altera_{Nome} : \forall \alpha, \beta.(\alpha :: \langle Nome : \beta \rangle) \Rightarrow \alpha \to \beta \to \alpha$$
$$e$$
$$altera_{Idade} : \forall \alpha, \beta.(\alpha :: \langle Idade : \beta \rangle) \Rightarrow \alpha \to \beta \to \alpha$$

A aplicação $(altera_{Nome}\ (Nome\_Idade\ x\ y) z)$ *em que* x, z *têm tipo* γ *e* y *tem tipo* δ*, tem como resultado a estrutura:*

$$\{Nome = z, Idade = y\}$$

com tipo $\{Nome : \gamma, Idade : \delta\}$*.*

4.2.3 O sistema de restrições $\mathcal{R}$

Vimos que em HM($\mathcal{R}$) as variáveis de tipo nos esquemas de tipos são restringidas por uma *espécie* κ.

Estas restrições surgem na nossa implementação como restrições primitivas da forma $(\tau :: \kappa)$, onde τ é um tipo e κ é um *espécie*. O nosso sistema de restrições $\mathcal{R}$ tem um predicado de aridade 2 (::), que adiciona restrições primitivas da forma $(\tau :: \kappa)$ ao nosso sistema.

Restrições da forma $\tau :: U$, não aparecem em HM($\mathcal{R}$) pois consideramos que, num esquema de tipos da forma $\forall\bar{\alpha}.C \Rightarrow \tau$, uma variável de tipo $\alpha \in \bar{\alpha}$ que não aparece restringida em C está quantificada universalmente, e pode representar qualquer tipo.

Seja $rectype(R)$ $(R = [(l_1, \tau_1), \ldots, (l_n, \tau_n)])$ a representação do tipo $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$, e $labtype(L, \tau_i)$ a representação do *espécie* $\langle L : \tau_i \rangle$. As regras de normalização para $\mathcal{R}$ são as seguintes:

$$records1 \quad @ \quad rectype(R) :: labtype(L, \tau_i) \Longleftrightarrow member(L, R, \tau_i) \mid true.$$

$$records2 \quad @ \quad \tau :: labtype(L, \tau_1), \tau :: labtype(L, \tau_2) \Longrightarrow \tau_1 = \tau_2.$$

$$records3 \quad @ \quad rectype(R) :: labtype(L, \tau_2) \Longrightarrow member(L, R, \tau_1) \mid \tau_2 = \tau_1.$$

$$records4 \quad @ \quad exists(\alpha, (\alpha :: \kappa)) \Longleftrightarrow \; fv(\kappa, V), notin(\alpha, V) \mid true.$$

A primeira regra diz-nos que, para qualquer l_i, τ_i $i = 1 \ldots n$, é verdade que $\{l_1 : \tau_1, \ldots, l_n : \tau_n\} :: \langle l_i : \tau_i \rangle$.

Exemplo 4.2.4 *Para a estrutura com campo Nome e Idade com tipos α e β, respectivamente, é verdade:*

$$\{Nome : \alpha, Idade : \beta\} :: \langle Nome : \alpha \rangle$$
$$e$$
$$\{Nome : \alpha, Idade : \beta\} :: \langle Idade : \beta \rangle$$

As duas regras seguintes garantem que não temos sobrecarga para campos de estruturas. Ou seja, uma determinada estrutura não pode ter tipos distintos para um mesmo campo.

Exemplo 4.2.5 *Se para um determinado tipo τ temos:*

$$\tau :: \langle Nome : \alpha \rangle \ e \ \tau :: \langle Nome : \beta \rangle$$

então, $\alpha = \beta$.

A última regra deriva das outras regras. Pela primeira regra temos:

$$\vdash^e \{l : \tau\} :: \langle l : \tau \rangle$$

De uma forma geral temos:

$$[\tau'/\alpha]C \vdash^e \exists\alpha.C$$

para qualquer restrição C, tipo τ' e variável α, pois:

$$\begin{array}{rcl} \tau' = \alpha \wedge C & \vdash^e & C \\ \exists\alpha.(\tau' = \alpha \wedge C) & \vdash^e & \exists\alpha.C \\ [\tau'/\alpha]C & \vdash^e & \exists\alpha.C \end{array}$$

Sendo $\{l : \tau\} :: \langle l : \tau \rangle$ a restrição $[\{l : \tau\}/\alpha]\, \alpha :: \langle l : \tau \rangle$, então:

$$\vdash^e \exists\alpha.\alpha :: \langle l : \tau \rangle$$

O teste $fv(\kappa, V), notin(\alpha, V)$ verifica que $\alpha \notin fv(\kappa)$, evitando estruturas recursivas.

4.2.4 O conjunto $\mathcal{S}$

Como foi definido em [Sul00], as restrições em forma resolvida são todas as restrições satisfazíveis da forma:

$$C ::= \text{true} \mid (\alpha :: \langle l : \tau \rangle) \mid C \wedge C \mid \exists\bar{\alpha}.C$$

Outra condição imposta em [Sul00] às restrições em $\mathcal{S}$, é que estejam em forma simplificada, isto é, $C \vdash^e (\tau = \tau')$ deve implicar $\vdash^e (\tau = \tau')$. Por exemplo,

$$(\alpha :: \langle l : \beta \rangle) \wedge (\alpha :: \langle l : \gamma \to \gamma \rangle)$$

não está em forma simplificada. Ao utilizarmos CHR na implementação das regras de normalização estamos a garantir a verificação dessa condição pois, sempre que é possível a aplicação de uma regra, essa aplicação é efectuada. Ou seja, no caso da restrição acima, a variável β é unificada com $\gamma \to \gamma$ resultando numa única restrição $(\alpha :: \langle l : \gamma \to \gamma \rangle)$.

4.2.5 Exemplos

Consideremos os seguintes exemplos de tipagens no sistema HM($\mathcal{R}$).

Exemplo 4.2.6 *Consideremos o seguinte programa*

$$\begin{array}{l} f\ x\ =\ \textbf{let} \\ \qquad\qquad g\ x\ y\ =\ \{l_1\ =\ x,\ l_2\ =\ y\} \\ \qquad\quad \textbf{in} \\ \qquad\quad altera\ l_1\ (g\ 1\ 2)\ x \end{array}$$

Os tipos inferidos para f e g, com as declarações $\{1 : int, 2 : int\}$ no ambiente inicial Γ_0, são respectivamente:

$$\begin{array}{l} g\ ::\ \forall\,\alpha,\,\beta\,.\{\}\ \Rightarrow\ \alpha\ \rightarrow\ \beta\ \rightarrow \{l_1\ :\ \alpha,\ l_2\ :\ \beta\} \\ f\ ::\ int\ \rightarrow\ \{l_1\ :\ int,\ l_2\ :\ int\} \end{array}$$

Exemplo 4.2.7 *Consideremos a seguinte função :*

$$f\ x\ y\ =\ eq\ x.l\ y.l$$

O tipo inferido para f, com a declaração $\{eq : \forall\alpha.\{\} \Rightarrow \alpha \rightarrow \alpha \rightarrow bool\}$ em Γ_0, é:

$$f\ ::\forall\,\alpha,\,\beta,\,\gamma\,.(\alpha\ ::\ \langle\ l\ :\ \beta\rangle\ \wedge\ \gamma\ ::\ \langle\ l\ :\ \beta\,\rangle\,)\ \Rightarrow \alpha\ \rightarrow\ \gamma\ \rightarrow\ bool$$

Exemplo 4.2.8 *Consideremos o seguinte programa*

$$\begin{array}{l} f\ x\ =\ \textbf{let} \\ \qquad\qquad g\ y\ =\ eq\ y\ x.l \\ \qquad\quad \textbf{in} \\ \qquad\quad 1 \end{array}$$

Os tipos inferidos para g e f, com as declarações $\{eq : \forall\alpha.\{\} \Rightarrow \alpha \rightarrow \alpha \rightarrow bool, 1 : int\}$ no ambiente inicial Γ_0, são respectivamente:

$$\begin{array}{l} g\ ::\ \forall\,\beta.\,(\alpha\ ::\ \langle\ l\ :\ \beta\,\rangle)\ \Rightarrow \beta\ \rightarrow\ bool \\ f\ ::\ \forall\,\alpha.\,(\exists\,\beta.\,(\alpha\ ::\ \langle\ l\ :\ \beta\rangle))\ \Rightarrow\ \alpha\ \rightarrow\ int \end{array}$$

Capítulo 5

Sobrecarga

No capítulo 2, quando apresentámos o sistema de tipos de Damas-Milner [DM82], falámos de polimorfismo paramétrico, onde as funções polimórficas funcionam uniformemente num conjunto de tipos.

Neste capítulo vamos falar de um outro tipo de polimorfismo, a sobrecarga de operadores. Como exemplo consideremos a função de igualdade (==). Embora seja desejável utilizarmos sempre (==) para indicar igualdade entre dois objectos, a forma como essa igualdade é implementada varia de objecto para objecto.

Com sobrecarga de operadores, o mesmo nome é usado para indicar funções diferentes, e o contexto em que o nome é usado permite determinar que função particular utilizar. Ou seja, a sobrecarga de operadores é uma forma puramente sintática de utilizar o mesmo nome para diferentes objectos semânticos.

Vamos descrever dois sistemas de tipos com sobrecarga de operadores: o sistema de tipos com classes de Nipkow e Prehofer [NP93], e o sistema O [OWW95].

5.1 Classes de Tipos

As classes de tipos permitem ter sobrecarga de funções, mantendo a vantagem dos tipos das expressões poderem ser inferidos automaticamente.

Uma classe de tipos é um conjunto de tipos. Consideremos novamente a função de igualdade. Num sistema sem classes poderíamos inferir o tipo polimórfico $\forall\alpha.\alpha \rightarrow \alpha \rightarrow bool$ para (==). No entanto, a função de igualdade não funciona uniformente para todos os tipos. A variável α não pode tomar como valor tipos funcionais, pois não é possível comparar funções. Além disso, a igualdade entre listas é implementada de maneira diferente da igualdade entre inteiros.

Com classes, uma variável de tipo α é restringida a elementos de uma determinada classe de tipos. Por exemplo, para cada tipo τ, para o qual a igualdade (==) está implementada, declaramos que τ pertence à classe Eq apresentando uma implementação de (==) com tipo $\tau \rightarrow \tau \rightarrow bool$.

Notação: Escrevemos $\tau : C$ para exprimirmos o facto do tipo τ pertencer à classe C. Podemos ver $\{C_1, \ldots, C_n\}$ como notação para $C_1 \cap \cdots \cap C_n$, a intersecção dos tipos que pertencem às classes C_1 a C_n. Assim, $\{C\}$ e C são equivalentes, e o conjunto vazio $\{\}$ representa o conjunto de todos os tipos.

Agora que já apresentamos de uma forma informal a noção de classe de tipos, passamos a descrever o sistema de tipos de Nipkow e Prehofer [NP93]. Com base nesse sistema apresentamos uma instância do HM($\mathcal{X}$), com classes definindo um sistema de restrições $\mathcal{TC}$, e o conjunto de regras de normalização de restrições nesse sistema.

5.1.1 O Sistema com Classes de Nipkow e Prehofer

5.1.1.1 Tipos e Classes

Definimos a gramática para os tipos, separando os tipos monomórficos τ dos tipos polimórficos σ:

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \tau \rightarrow \tau' \mid t(\tau_n, \ldots, \tau_n) \\ \sigma & ::= & \tau \mid \forall \alpha : S.\sigma \end{array}$$

O termo $t(\tau_1, \ldots, \tau_n)$ representa um construtor de tipo de aridade $n \geq 0$. Note-se que $\rightarrow$ é também um construtor de tipo, de aridade 2, isto é, podemos ver $\tau_1 \rightarrow \tau_2$ como uma representação de $\rightarrow (\tau_1, \tau_2)$.

Exemplo 5.1.1 *Tipos básicos como **int**, **float** e **strint** são exemplos de construtores de tipo, de aridade zero.*

Como exemplo de um construtor de tipo de aridade maior que zero consideremos o seguinte:

$$\boldsymbol{par}(int, int)$$

As variáveis de tipo nos esquemas de tipos, aparecem restringidas a um conjunto finito de classes $S = \{C_1, \ldots, C_n\}$. Sendo $\{C_1, \ldots, C_n\}$ equivalente a $C_1 \cap \cdots \cap C_n$, dizer que um tipo τ pertence a $\{C_1, \ldots, C_n\}$, notação $\tau : S$, é equivalente a $\tau \in C_1 \cap \cdots \cap C_n$ ou ainda $\tau : C_1 \wedge \cdots \wedge \tau : C_n$.

5.1.1.2 Classes e Instâncias

Uma hierarquia de classes define uma relação de ordem entre classes ($\leq$).

Definição 5.1.1 *Definimos uma hierarquia de classes através de declarações da forma:*

$$C_1 \leq C_2$$

onde C_1 e C_2 são duas classes de tipos, e $\leq$ é uma relação de ordem parcial entre classes. Na declaração acima, dizemos que C_1 é subclasse de C_2, e que C_2 é superclasse de C_1.

Definição 5.1.2 *Sejam C_1 e C_2 duas classes de tipos. Se $C_1 \leq C_2$, ou seja C_1 é subclasse de C_2 (C_2 superclasse de C_1), então $\tau : C_1$ implica que $\tau : C_2$.*

A relação entre classes é não cíclica, isto é deve formar um grafo acíclico dirigido (dag).

Uma declaração de classe é um termo da forma:

$$\textbf{class } \alpha : C \leq \{C_1, \ldots, C_n\} \textbf{ where } x : \sigma$$

tal que $C_1, \ldots, C_n$ são superclasses de C ($C \leq C_i$, $i = 1 \ldots n$), e x é uma função definida em C, em que a única variável livre em σ é α. A declaração acima corresponde em Haskell à declaração:

$$\textbf{class } (C_1\ \alpha, \ldots, C_n\ \alpha) \Rightarrow C\ \alpha \textbf{ where } x :: \tau$$

onde τ é o corpo de σ.

Exemplo 5.1.2 *A seguinte declaração de classe*

$$\textbf{class } \alpha : Eq \textbf{ where } eq : \alpha \to \alpha \to bool$$

define a classe Eq de todos os tipos para os quais a função $eq : \alpha \to \alpha \to bool$ está definida. Uma declaração de classe é como um módulo de interface: separa declarações de implementações.

Uma declaração de classe **class** $\alpha : C \leq S$ **where** $x : \sigma$ tem que ser precedida por uma declaração para cada superclasse em S, mas não por outra declaração **class** $\alpha : C$, ou seja, a relação de superclasse não pode ser cíclica.

Uma declaração de instância permite declarar que determinado tipo ou construtor de tipo pertence a uma dada classe. Para isso é necessário apresentar implementações das funções declaradas nas classes que respeitem a declaração de tipo dada. A sintaxe para declarações de instâncias é a seguinte:

$$\mathbf{inst}\ t : (S_1, \ldots, S_n)C\ \mathbf{where}\ x = M$$

e significa que $t(\tau_1, \ldots, \tau_n)$ pertence à classe C desde que τ_i seja do tipo S_i. Corresponde em Haskell à declaração:

$$\mathbf{inst}\ (con) \Rightarrow C(t\ \alpha_1 \ldots \alpha_n)\ \mathbf{where}\ x = M$$

onde *con* é uma lista de hipóteses $C'\alpha_i$ com $C' \in S_i$, para $i = 1 \ldots n$.

Um tipo não pode ser declarado como instância de uma determinada classe mais do que uma vez no mesmo programa.

Exemplo 5.1.3 *Consideremos a seguinte declaração de instância:*

$$\mathbf{inst}\ int : Eq\ \mathbf{where}\ eq = eq\_int$$

A função eq_int é uma função existente de tipo $eq : int \to int \to bool$.

5.1.1.3 Equivalência entre Conjuntos de Classes

As declarações de classe definem uma hierarquia de classes, descrita pela relação de ordem entre classes $\leq$. Com base na relação $\leq$ definimos uma relação entre conjuntos de classes, $\preceq$.

Definição 5.1.3 *Dados dois conjuntos de classes* S_1 *e* S_2*, escrevemos* $S_1 \preceq S_2$ *quando* S_1 *é mais especializado, ou seja, representa menos tipos que* S_2*. Dada uma ordem parcial* $\leq$ *nas classes, a relação* $\preceq$ *é definida por:*

$$S_1 \preceq S_2 \iff \forall C_2 \in S_2 . \exists C_1 \in S_1 . C_1 \leq C_2$$

É fácil verificar que se $S_2 \subseteq S_1$, então $S_1 \preceq S_2$.

Exemplo 5.1.4 *Seja* $S_1 = \{Ord\}$, *e* $S_2 = \{Ord, Eq\}$. *Como* $\{Ord\} \subseteq \{Ord, Eq\}$, *então*

$$\{Ord, Eq\} \preceq \{Ord\}$$

Definimos uma equivalência entre conjuntos de classes $\approx$, no sentido em que, se $S_1 \approx S_2$, então S_1 e S_2 representam os mesmos tipos, da seguinte forma:

Definição 5.1.4 *Sejam* S_1 *e* S_2 *dois conjuntos de tipos, escrevemos* $S_1 \approx S_2$ *e dizemos que* S_1 *e* S_2 *são equivalentes se e só se:*

$$S_1 \approx S_2 \Leftrightarrow S_1 \preceq S_2 \wedge S_2 \preceq S_1$$

Exemplo 5.1.5 *Para* S_1 *e* S_2 *do exemplo anterior, vimos que* $S_2 \preceq S_1$. *Supondo que:*

$$Ord \leq Ord \wedge Ord \leq Eq$$

temos $\{Ord\} \preceq \{Ord, Eq\}$, *e portanto*

$$\{Ord\} \approx \{Ord, Eq\}$$

ou seja, S_1 *e* S_2 *representam o mesmo conjunto de tipos.*

5.1.1.4 Linguagem de termos

Um programa neste sistema é uma sequência de declarações seguida de uma expressão. Assim, definimos a seguinte gramática para programas:

$$M \quad = \quad x \mid M\,M \mid \lambda x.M \mid \text{let } x = M \text{ in } M$$

$$\begin{array}{rcl} P & = & M \mid \textbf{class } \alpha : C \leq S \textbf{ where } x : \sigma;\ P \\ & \mid & \textbf{inst } t : (S_1, \ldots, S_n)C \textbf{ where } x = M;\ P \end{array}$$

Exemplo 5.1.6 *Um programa na linguagem acima,é por exemplo:*

> **class** α : *Num* **where** $+ \ : \ \alpha \ \rightarrow \ \alpha \ \rightarrow \ \alpha;$
> **inst** *int* : *Num* **where** $+ \ = \ plus\_int;$
> $incr\ x \ = \ x \ + \ 1$

onde plus_int é a função de soma nos inteiros com tipo $int \rightarrow int \rightarrow int$

5.1.1.5 Regras de Inferência

Neste sistema, para além de declarações da forma $M : \tau$, que classificam um termo como sendo do tipo τ, existem também declarações da forma $\tau : S$, que classificam um tipo com pertencendo ao conjunto de classes S.

Definição 5.1.5 *Seja α uma variável de tipo e S um conjunto finito de classes. Temos que:*

- *Um ambiente $\mathcal{S}$ é um conjunto de declarações da forma $\alpha : S$.*
- *O domínio de um ambiente $\mathcal{S} = \{\alpha_1 : S_1, \ldots, \alpha_n : S_n\}$, $dom(\mathcal{S})$, é $\{\alpha_1, \ldots, \alpha_n\}$, e $S_i = \mathcal{S}(\alpha_i)$.*

A partir das declarações de instâncias é construído um conjunto Δ de declarações da forma $t : (\overline{S_n})\ C$, que determinam a funcionalidade dos construtores de tipo.

As regras seguintes, para declarações de classe e instância, definem os valores iniciais de Δ, $\leq$, e Γ. As declarações de classe definem Γ e $\leq$, e as declarações de instância definem Δ.

Ao par $\Sigma = (\Delta, \leq)$, é chamado assinatura de tipos.

A regra para classes acrescenta à relação de ordem entre classes $\leq$, o fecho transitivo $(\leq \cup \{(C, D) \mid D \in S\})^*$ de $\leq$ e a nova relação de subclasse declarada.

Seja Σ uma assinatura de classes, $\mathcal{S}$ um ambiente, Γ uma base, M um termo, e σ um esquema de tipos. Escrevemos $\Sigma, \mathcal{S}, \Gamma \vdash M : \sigma$ se obtemos $M : \sigma$ pela aplicação das seguintes regras:

$$(Class) \quad \frac{\Delta, (\leq \cup \{(C, D) \mid D \in S\})^*, \mathcal{S}, \Gamma \cup \{x : \forall \alpha : C.\sigma\} \vdash P : \sigma'}{\Delta, \leq, \mathcal{S}, \Gamma \vdash \textbf{class}\ \alpha : C \leq S\ \textbf{where}\ x : \sigma; P : \sigma'}$$

$$(Inst) \quad \frac{\begin{array}{c}\Delta \cup \{t : (\overline{S_n})S\}, \leq, \mathcal{S}, \Gamma \vdash P : \sigma' \\ \Gamma(x) = \forall \alpha : C.\sigma \qquad \Delta, \leq, \mathcal{S} \cup \{\overline{\alpha_n : S_n}\}, \Gamma \vdash M : [t(\overline{\alpha_n})/\alpha]\sigma\end{array}}{\Delta, \leq, \mathcal{S}, \Gamma \vdash \textbf{inst}\ t : (\overline{S_n})C\ \textbf{where}\ x = M; P : \sigma'}$$

Exemplo 5.1.7 *Consideremos as seguintes declarações de classe e instância:*

$$\begin{array}{ll}
(C_1) & \mathbf{class}\ \alpha : Eq\ \mathbf{where}\ eq : \alpha \to \alpha \to bool; \\
(C_2) & \mathbf{class}\ \alpha : Ord \leq Eq\ \mathbf{where}\ le : \alpha \to \alpha \to bool; \\
(I_1) & \mathbf{inst}\ int : Eq\ \mathbf{where}\ eq = eq_int; \\
(I_2) & \mathbf{inst}\ par : (Eq, Eq)Eq\ \mathbf{where}\ eq = eq_par;
\end{array}$$

A partir das declarações acima, aplicando as regras de classe e instância obtemos:

$$\Delta = \{\},\ \ \leq = \{\},\ \ \Gamma = \{\}$$

$$\Big\downarrow \ \mathit{Class}\ (C_1)$$

$$\begin{array}{lcl}
\Delta & = & \{\},\ \ \leq\ =\ \{\} \\
\Gamma & = & \{eq : \forall\alpha : Eq.\alpha \to \alpha \to bool\}
\end{array}$$

$$\Big\downarrow \ \mathit{Class}\ (C_2)$$

$$\begin{array}{lcl}
\Delta & = & \{\},\ \ \leq\ =\ \{(Ord, Eq)\} \\
\Gamma & = & \{eq : \forall\alpha : Eq.\alpha \to \alpha \to bool,\ le : \forall\alpha : Ord.\alpha \to \alpha \to bool\}
\end{array}$$

$$\Big\downarrow \ \mathit{Inst}\ (I_1)$$

$$\begin{array}{lcl}
\Delta & = & \{int : Eq\} \\
\leq & = & \{(Ord, Eq)\} \\
\Gamma & = & \{eq : \forall\alpha : Eq.\alpha \to \alpha \to bool,\ le : \forall\alpha : Ord.\alpha \to \alpha \to bool\}
\end{array}$$

$$\Big\downarrow \ \mathit{Inst}\ (I_2)$$

$$\begin{array}{lcl}
\Delta & = & \{int : Eq,\ par : (Eq, Eq)Eq\} \\
\leq & = & \{(Ord, Eq)\} \\
\Gamma & = & \{eq : \forall\alpha : Eq.\alpha \to \alpha \to bool,\ le : \forall\alpha : Ord.\alpha \to \alpha \to bool\}
\end{array}$$

Determinar que um tipo é restringido por um conjunto de classes depende:

- Das classes associadas às variáveis de tipo em τ, que aparecem declarados num ambiente $\mathcal{S}$;
- Do comportamento dos construtores de tipo, declarado em Δ;
- Da relação de ordem entre classes $\leq$.

Dados Σ uma assinatura de tipos, $\mathcal{S}$ um ambiente, τ um tipo, e S um conjunto de classes, o tipo τ é restringido por S, escrevemos $\Sigma, \mathcal{S} \vdash \tau : S$, se $\tau : S$ pode ser obtido pela aplicação das seguintes regras:

$$(1) \qquad \frac{\Sigma, \mathcal{S} \vdash \tau : C_i \quad i = 1 \ldots n}{\Sigma, \mathcal{S} \vdash \tau : \{C_1, \ldots, C_n\}}$$

$$(2) \qquad \frac{\Sigma, \mathcal{S} \vdash \tau : \{C_1, \ldots, C_n\}}{\Sigma, \mathcal{S} \vdash \tau : C_i \quad i = 1 \ldots n}$$

$$(3) \qquad \frac{\mathcal{S}(\alpha) = S}{\Sigma, \mathcal{S} \vdash \alpha : S}$$

$$(4) \qquad \frac{t : (\overline{S_n})C \in \Delta \qquad \Sigma, \mathcal{S} \vdash \tau_i : S_i \quad i = 1 \ldots n}{\Sigma, \mathcal{S} \vdash t(\overline{\tau_n}) : C}$$

$$(5) \qquad \frac{\Sigma, \mathcal{S} \vdash \tau : C_1 \qquad C_1 \leq C_2}{\Sigma, \mathcal{S} \vdash \tau : C_2}$$

Exemplo 5.1.8 *Consideremos as seguintes declarações:*

> **class** α : Eq **where** $eq : \alpha \to \alpha \to bool$;
> **class** α : $Ord \leq Eq$ **where** $le : \alpha \to \alpha \to bool$;
> **inst** int : Ord **where** $le = le\_int$

onde le_int define a relação de menor ou igual nos inteiros.

A partir das declarações acima construímos $\Delta = \{int : Ord\}$, $\leq = \{(Ord, Eq)\}$ e $\mathcal{S} = \{\}$. Temos então que:

$$\begin{tikzcd}
 & & int : Ord \in \Delta \arrow[d, "(4)"] & & \\
int : Ord \in \Delta \arrow[d, "(4)"'] & & \Sigma, \mathcal{S} \vdash int : Ord \arrow[dr, "(5)"'] & & Ord \leq Eq \arrow[dl, "(5)"] \\
\Sigma, \mathcal{S} \vdash Int : Ord \arrow[drr, "(1)"'] & & & \Sigma, \mathcal{S} \vdash int : Eq \arrow[dl, "(1)"] & \\
 & & \Sigma, \mathcal{S} \vdash int : \{Ord, Eq\} & &
\end{tikzcd}$$

A construção de Δ e $\leq$ é feita aplicando as regras de declaração de classes e instância.

Inferência de tipos Seja Σ uma assinatura de classes, $\mathcal{S}$ um ambiente, Γ uma base, M um termo, e σ um esquema de tipos. No sistema de Nipkow e Prehofer, escreve-se $\Sigma, \mathcal{S}, \Gamma \vdash M : \sigma$ se obtemos $M : \sigma$ pela aplicação das seguintes regras:

$$(Ax) \qquad \Sigma, \mathcal{S}, \Gamma \vdash x : \Gamma(x)$$

$$(\forall E) \qquad \frac{\Sigma, \mathcal{S}, \Gamma \vdash M : \forall\alpha : S.\sigma \qquad \Sigma, \mathcal{S} \vdash \tau : S}{\Sigma, \mathcal{S}, \Gamma \vdash M : [\tau/\alpha]\sigma}$$

$$(\forall I) \qquad \frac{\Sigma, \mathcal{S} \cup \{\alpha : S\}, \Gamma \vdash M : \sigma \qquad \alpha \in FV(\sigma) \setminus FV(\Gamma)}{\Sigma, \mathcal{S}, \Gamma \vdash M : \forall\alpha : S.\sigma}$$

$$(Apl) \qquad \frac{\Sigma, \mathcal{S}, \Gamma \vdash M_1 : (\tau' \to \tau) \qquad \Sigma, \mathcal{S}, \Gamma \vdash M' : \tau'}{\Sigma, \mathcal{S}, \Gamma \vdash (MM') : \tau}$$

$$(Abs) \qquad \frac{\Sigma, \mathcal{S}, \Gamma \cup \{x : \tau'\} \vdash M : \tau}{\Sigma, \mathcal{S}, \Gamma \vdash \lambda x.M : (\tau' \to \tau)}$$

$$(Let) \qquad \frac{\Sigma, \mathcal{S}, \Gamma \vdash M : \sigma \qquad \Sigma, \mathcal{S}, \Gamma \cup \{x : \sigma\} \vdash M' : \sigma'}{\Sigma, \mathcal{S}, \Gamma \vdash \text{let } x = M \text{ in } M' : \sigma'}$$

Exemplo 5.1.9 *Consideremos o seguinte programa:*

class α : *Num* **where** $+ \; : \; \alpha \to \alpha \to \alpha;$
inst *int* : *Num* **where** $+ \; = \; plus\_int$
$f \; = \; + \, x \, y$

Supondo que x e y são inteiros construímos $\Gamma = \{+ : \forall\alpha : Num.\alpha \to \alpha \to \alpha,\ x : int,\ y : int\}$ e $\Delta = \{int : Num\}$, aplicando as regras Class e Inst. Para a expressão $(+\ x\ y)$, temos a seguinte derivação:

$$\begin{tikzcd}
\Sigma, \mathcal{S}, \Gamma \vdash + : \forall\alpha : Num.\alpha \to \alpha \to \alpha \arrow[dr, "\forall E"'] & & \Sigma, \mathcal{S} \vdash int : Num \arrow[dl, "\forall E"] & & \\
& \Sigma, \mathcal{S}, \Gamma \vdash + : int \to int \to int \arrow[dr, "APP"'] & & \Sigma, \mathcal{S}, \Gamma \vdash x : int \arrow[dl, "APP"] & \\
& & \Sigma, \mathcal{S}, \Gamma \vdash +\ x : int \to int \arrow[dr, "APP"'] & & \Sigma, \mathcal{S}, \Gamma \vdash y : int \arrow[dl, "APP"] \\
& & & \Sigma, \mathcal{S}, \Gamma \vdash +\ x\ y : int &
\end{tikzcd}$$

5.1.1.6 As classes do Haskell e o sistema de Nipkow e Prehofer

O sistema de Nipkow e Prehofer define o sistema de classes de tipos da linguagem Haskell, a menos da simplificação de classes. Consideremos o seguinte programa:

$$\begin{array}{l}
\textbf{class}\ \alpha\ :\ Eq\ \textbf{where}\ eq\ :\ \alpha \to \alpha \to bool;\\
\textbf{class}\ \alpha\ :\ Ord\ \leq\ Eq\ \textbf{where}\ le\ :\ \alpha \to \alpha \to bool;\\
g\ x\ y\ =\ le\ x\ y
\end{array}$$

O tipo inferido para g neste sistema é:

$$g\ ::\ \forall\,\alpha.\ \alpha\ :\ \{Eq, Ord\}\ \Rightarrow\ \alpha \to \alpha \to bool$$

Com base na relação de equivalência entre tipos temos $\{Eq, Ord\} \approx \{Ord\}$. Assim $\alpha : \{Eq, Ord\}$ é equivalente a $\alpha : \{Ord\}$ e o tipo acima pode ser simplificado para:

$$g\ ::\ \forall\,\alpha.\ \alpha\ :\ \{Ord\}\ \Rightarrow\ \alpha \to \alpha \to bool$$

Esta operação de simplificação está presente no sistema de classes da linguagem Haskell, mas não é feita no sistema de Nipkow e Prehofer.

5.1.2 Mundo Aberto *versus* Mundo Fechado

Antes de apresentarmos a implementação da inferência de tipos para o sistema de Nipkow e Prehofer, vamos distinguir sistemas *mundo aberto* de sistemas *mundo fechado*.

Numa implementação *mundo aberto* de um sistema é possível a existência de declarações de tipos para funções que não estão implementadas. Por exemplo, num sistema com classes significa que podemos declarar tipos para métodos das classes, sem no entanto apresentarmos instâncias desses métodos.

Num *mundo fechado* é necessário termos conhecimento de todos os factos no momento em que fazemos inferência de tipos. Por exemplo, num sistema com classes é necessário apresentar instâncias de todos os métodos declarados.

Em termos de implementação, a decisão de mundo aberto/mundo fechado é determinada pela forma como a projecção é definida nas restrições.

Num contexto de variáveis de tipo restringidas por classes (como nas classes do Haskell), consideremos uma restrição da forma $\exists\alpha.\alpha : Eq$. Podemos especificar uma implementação mundo aberto com uma regra nas restrições da forma:

$$\exists\alpha.\alpha : Eq =^e \mathit{true}$$

pois, uma vez que não necessitamos de ter instâncias para as classes que declaramos, assumimos como verdadeira a restrição $\exists\alpha.\alpha : Eq$.

Numa implementação mundo fechado, a restrição acima só é satisfazível se existir pelo menos uma instância da classe Eq. Imaginemos que temos uma instância da classe Eq nos inteiros. Uma implementação mundo fechado pode ser especificada pelas seguintes regras nas restrições:

$$\begin{array}{rcl} \mathit{true} & \vdash^e & \mathit{Int} : Eq \\ \mathit{Int} : Eq & \vdash^e & \exists\alpha.\alpha : Eq \end{array}$$

Os sistemas de restrições que vamos apresentar neste capítulo optam sempre por um sistema mundo aberto.

5.1.3 O HM($\mathcal{TC}$)

Apresentamos agora uma implementação de inferência de tipos para o sistema de Nipkow e Prehofer com classes de tipos, definindo o sistema de restrições $\mathcal{TC}$. Esta implementação é baseada em [GSM00].

5.1.3.1 A linguagem de tipos $\mathcal{T}$

A linguagem de tipos $\mathcal{T}$ para o sistema de restrições $\mathcal{TC}$ pela seguinte gramática:

$$\tau \quad ::= \quad \alpha \mid \tau \to \tau' \mid t(\tau_1, \ldots, \tau_n)$$

Para além de variáveis de tipo e tipos funcionais, $t(\tau_1, \ldots, \tau_n)$ representa construtores de tipo de aridade n ($n \geq 0$).

5.1.3.2 A linguagem de termos e o ambiente inicial Γ_0

A linguagem de termos para o sistema de Nipkow e Prehofer, extende a linguagem do sistema de Damas-Milner com declarações de classes e instâncias.

A linguagem de termos é a mesma do sistema de Nipkow e Prehofer. As declarações de classe acrescentam ao ambiente inicial as declarações de tipo dos operadores com sobrecarga.

Assim, para cada declaração de classe

$$\mathbf{class}\ \alpha : C \leq S\ \mathbf{where}\ x : \sigma$$

acrescentamos ao ambiente inicial Γ_0 a declaração de tipo $x : \sigma$, para o operador com sobrecarga x.

Exemplo 5.1.10 *Consideremos o seguinte programa:*

$$\begin{array}{l}
\mathbf{class}\ \alpha\ :\ Eq\ \mathbf{where}\ eq\ :\ \alpha\ \rightarrow\ \alpha\ \rightarrow\ bool; \\
\mathbf{class}\ \alpha\ :\ Ord\ \leq\ Eq\ \mathbf{where}\ le\ :\ \alpha\ \rightarrow\ \alpha\ \rightarrow\ bool; \\
f\ x\ y\ =\ eq\ x\ y \\
g\ x\ y\ =\ le\ x\ y
\end{array}$$

O ambiente inicial para o processo de inferência de f e g é

$$\Gamma_0 = \{eq : \alpha \rightarrow \alpha \rightarrow bool,\ le : \alpha \rightarrow \alpha \rightarrow bool\}$$

5.1.3.3 O sistema de restrições $\mathcal{TC}$

As variáveis de tipo nos esquemas de tipos podem aparecer restringidas por uma classe C.

Isso corresponde a termos restrições primitivas da forma $(\alpha :: C)$, onde α é uma variável de tipo e C é uma classe. O nosso sistema de restrições $\mathcal{R}$ tem um predicado de aridade 2 (::), que adiciona restrições primitivas da forma $(\alpha :: C)$ ao nosso sistema.

Sendo $S = \{C_1, \ldots, C_n\}$ um conjunto de classes, então uma restrição da forma $\tau :: S$ é equivalente a uma restrição da forma $(\tau :: C_1) \wedge \cdots \wedge (\tau :: C_n)$.

Em $\mathcal{TC}$ as restrições primitivas são sempre da forma $\tau :: C$, sendo C uma classe de tipos, e restrições da forma $\tau :: S$ são transformadas na restrição equivalente $(\tau :: C_1) \wedge \cdots \wedge (\tau :: C_n)$.

Seja $\tau :: [C_1, \ldots, C_n]$ a representação de $\tau :: \{C_1, \ldots, C_n\}$. As regras de normalização para $\mathcal{TC}$ são as seguintes:

$$\begin{array}{lcl} class1 & @ & (\alpha :: [C]) \Longleftrightarrow (\alpha :: C). \\ \\ class2 & @ & (\alpha :: [C_1 \mid RC]) \Longleftrightarrow (\alpha :: C_1),\ (\alpha :: RC). \\ \\ class3 & @ & exists(\alpha, S) \Longleftrightarrow classes(\alpha, S) \mid true. \end{array}$$

As duas primeiras regras mantêm a equivalência entre restrições da forma $\alpha :: \{C_1, \ldots, C_n\}$ e restrições da forma $(\alpha :: C_1) \wedge \cdots \wedge (\alpha :: C_n)$.

A última regra implementa um sistema *mundo aberto*. O predicado $classes(\alpha, S)$ verifica que S é conjunção de restrições da forma $(\alpha :: C_1) \wedge \cdots \wedge (\alpha :: C_n)$, que é equivalente a uma restrição da forma $\alpha :: S'$, onde $S' = \{C_1, \ldots, C_n\}$.

A linguagem de termos tem dois termos adicionais para declaração de classes e instâncias. No sistema de Nipkow e Prehofer vimos que as regras de *Classe* e *Instância*, construíam o ambiente Γ, e a assinatura de tipos Δ.

Assumindo que as declarações de classe e instância estão disponíveis antes do processo de inferência, então numa fase de pré-processamento, percorremos as declarações construindo o ambiente inicial Γ_0, e acrescentando novas regras ao sistema de restrições de modo a ter em conta a hierarquia de classes e o funcionamento dos construtores de tipo.

As regras de pré-processamento são as seguintes.

$$\begin{array}{rl} pp(\Gamma, \mathbf{class}\ \alpha : C \leq S\ \mathbf{where}\ x : \sigma; P) \leftarrow & add\_class\_rule(\alpha, C, S), \\ & pp([(x, \sigma) \mid \Gamma], P). \end{array} \qquad \text{(Class)}$$

$$\begin{array}{rl} pp(\Gamma, \mathbf{inst}\ t : (\overline{S})\, C\ \mathbf{where}\ x = M; P) \leftarrow & member(x, \Gamma, \forall\alpha.(\alpha :: C) \Rightarrow \sigma), \\ & type(C, \Gamma, M, [t\bar{\alpha}/\alpha]\sigma), \\ & add\_inst\_rule(\bar{\alpha}, t, \overline{S}, C), \\ & pp(\Gamma, P). \end{array} \qquad \text{(Inst)}$$

Sendo C uma classe e $S = [C_1, \ldots, C_n]$, a função $add\_class\_rule(\alpha, C, S)$ acrecenta uma nova regra de normalização ao sistema $\mathcal{TC}$ que lida com a relação de ordem entre as classes, da forma:

$$RC \ @ \ (\alpha :: C) \Longrightarrow (\alpha :: C_1) \wedge \cdots \wedge (\alpha :: C_n)$$

Seja $\bar{\alpha} = [\alpha_1, \ldots, \alpha_n]$, $\overline{S} = [S_1, \ldots, S_n]$, C uma classe, e t um construtor de tipo. A função $add\_inst\_rule(\bar{\alpha}, t, \overline{S}, C)$, acrescenta ao sistema de restrições uma regra da forma:

$$RI \ @ \ (t\,\alpha_1 \ldots \alpha_n :: C) \Longleftrightarrow (\alpha_1 :: S_1) \wedge \cdots \wedge (\alpha_n :: S_n)$$

que permite lidar com construtores de tipo.

À semelhança do que acontecia no sistema de classes de tipos de Nipkow e Prehofer, as declarações de classe extendem o ambiente inicial e a relação de ordem entre as classes, e as declarações de instância determinam o comportamento dos construtores de tipo.

5.1.3.4 O conjunto de restrições em forma resolvida $\mathcal{S}$

O conjunto $\mathcal{S}$ de restrições em forma resolvida consiste em todas as restrições satisfazíveis da forma:

$$C ::= \{\} \mid (\alpha :: C) \mid C \wedge C$$

A regra $class3$ do sistema de restrições define a projecção como sendo trivial para restrições da forma $(\alpha :: S)$. As restrições primitivas são sempre dessa forma, o que implica que a projecção seja sempre trivial. Assim, não existem no conjunto $\mathcal{S}$, restrições da forma $\exists\alpha.C$

5.1.4 Exemplos

Exemplo 5.1.11 *Consideremos o seguinte programa:*

> **class** α : Num **where** $+ \ : \ \alpha \ \rightarrow \ \alpha \ \rightarrow \ \alpha$;
> **inst** int : Num **where** $+ \ = \ plus\_int$
> $f\ x\ y \ = \ x \ + \ y$
> $g\ x \ = \ x \ + \ 1$

Sendo $plus\_int$ a função de soma nos inteiros. Os tipos inferidos para f e g com ambiente *inicial* $\Gamma_0 = \{1 : int\}$ *são:*

$$f \;::\; \forall\, \alpha.\, \alpha \;::\; Num \;\Rightarrow\; \alpha \;\rightarrow\; \alpha \;\rightarrow\; \alpha$$
$$g \;::\; int \;\rightarrow\; int$$

Exemplo 5.1.12 *Para o programa:*

$$\mathbf{class}\ \alpha\ :\ Num\ \mathbf{where}\ +\ :\ \alpha\ \rightarrow\ \alpha\ \rightarrow\ \alpha;$$
$$\mathbf{inst}\ int\ :\ Num\ \mathbf{where}\ +\ =\ plus\_int$$
$$\mathbf{inst}\ float\ :\ Num\ \mathbf{where}\ +\ =\ plus\_float$$
$$f\ x\ =\ x\ +\ toInt(2.4\ +\ 2.5)$$

Onde plus_int e plus_float são a soma para inteiros e vírgula flutuante, respectivamente. O tipo inferido para f*, com o ambiente inicial* $\Gamma_0 = \{toInt : \forall\alpha.\{\} \Rightarrow \alpha \rightarrow int, 2.4 : float, 2.5 : float\}$*, é:*

$$f \;::\; int \;\rightarrow\; int$$

Exemplo 5.1.13 *Para o programa:*

$$\mathbf{class}\ \alpha\ :\ Num\ \mathbf{where}\ +\ :\ \alpha\ \rightarrow\ \alpha\ \rightarrow\ \alpha;$$
$$\mathbf{inst}\ int\ :\ Num\ \mathbf{where}\ +\ =\ plus\_int$$
$$f\ x\ y\ =\ x\ +\ toInt(y)$$
$$g\ x\ y\ =\ toInt(x)\ +\ y$$
$$h\ x\ y\ =\ toInt(x)\ +\ toInt(y)$$

Os tipos inferidos para f*,* g *e* h*, com o ambiente inicial* $\Gamma_0 = \{toInt : \forall\alpha.\{\} \Rightarrow \alpha \rightarrow int\}$*, são:*

$$f \;::\; \forall\, \alpha.\, \{\} \;\Rightarrow\; int \;\rightarrow\; \alpha \;\rightarrow\; int$$
$$g \;::\; \forall\, \alpha.\, \{\} \;\Rightarrow\; \alpha \;\rightarrow\; int \;\rightarrow\; int$$
$$h \;::\; \forall\, \alpha,\, \beta.\, \{\} \;\Rightarrow\; \alpha \;\rightarrow\; \beta \;\rightarrow\; int$$

Exemplo 5.1.14 *Exemplo*

$$\mathbf{class}\ \alpha\ :\ Eq\ \mathbf{where}\ eq\ :\ \alpha\ \rightarrow\ \alpha\ \rightarrow\ bool;$$
$$\mathbf{class}\ \alpha\ :\ Ord\ \leq\ Eq\ \mathbf{where}\ le\ :\ \alpha\ \rightarrow\ \alpha\ \rightarrow\ bool;$$
$$f\ x\ y\ =\ eq\ x\ y$$
$$g\ x\ y\ =\ le\ x\ y$$

Os tipos inferidos para f e g com ambiente inicial $\Gamma_0 = \{\}$ são:

$$f \ :: \ \forall\, \alpha.\, \alpha \ : \ Eq \ \Rightarrow \ \alpha \ \rightarrow \ \alpha \ \rightarrow \ bool$$
$$g \ :: \ \forall\, \alpha.\, (\alpha \ : \ Eq) \ \wedge \ (\alpha \ : \ Ord) \ \Rightarrow \ \alpha \ \rightarrow \ \alpha \ \rightarrow \ bool$$

5.1.5 Ambiguidade

Um problema que afecta o sistema de tipos com classes é a ambiguidade. A causa da ambiguidade dos programas nestes sistemas, vem do facto de que, apesar de um programa poder ter um tipo único, a sua semântica pode não estar bem definida. Consideremos o seguinte programa:

class $\alpha \ : \ C$ **where** $f \ : \ \alpha \ \rightarrow \ int;$
class $\alpha \ : \ D$ **where** $c \ : \ \alpha;$
$(f\ c)$

Aplicando o algoritmo ao programa obtemos que o seu tipo é $\forall \alpha.(\alpha \ :: \ \{C, D\}) \ \Rightarrow \ int.$ No entanto, o programa não tem semântica pois não existem instâncias de f e c, e caso existissem múltiplas instâncias tanto de C como de D, seria impossível determinar quais usar na expressão $(f c)$.

Assim, diz-se que uma expressão e tem um tipo ambíguo se, no seu tipo $\forall \bar{\alpha}.C \Rightarrow \tau$, existe uma variável de tipo α que ocorre em C mas não ocorre em τ.

Consideremos o seguinte programa em Haskell:

class $(Show\ a)$ **where**
$show \ :: \ a \ \rightarrow \ String$
class $(Read\ a)$ **where**
$read \ :: \ String \ \rightarrow \ a$
$f \ = $ **let**
  $x \ = \ read$ "..."
 in
 $show\ x$

O tipo da função f é $\forall a.(Show\ a, Read\ a) \Rightarrow \ String.$

Considerando *Int* e *Bool* como sendo instâncias de *Read* e *Show*, a função f é ambígua, pois nos tipos para *show* e *read*, a variável de tipo a pode ser instanciada tanto a *Int* como a *Bool*.

5.2 Sobrecarga de operadores e o Sistema $\mathcal{O}$

Nesta secção, apresentamos o sistema $\mathcal{O}$ [OWW95] como instância do HM($\mathcal{X}$). O sistema $\mathcal{O}$ apresenta uma forma restrita de sobrecarga que resolve o problema da ambiguidade, de que falamos na secção anterior.

5.2.1 Sistema $\mathcal{O}$

Neste sistema, dada uma classe C sobre uma variável de tipo α, cada operador com sobrecarga em C deve ter um tipo da forma $\alpha \to \tau$, onde τ pode também envolver α. Ou seja, o operador com sobrecarga deve ser sempre do tipo funcional $(\alpha \to \tau)$ e a sobrecarga é feita sobre o primeiro argumento α.

Exemplo 5.2.1 *Consideremos as seguintes declarações de classe em Haskell:*

class (*Show a*) **where**
show :: $a \to$ *String*

class (*Num a*) **where**
$(+)$ :: $a \to a \to a$
fromInt :: *Int* $\to a$

class (*Read a*) **where**
read :: *String* $\to a$

As funções show da classe Show e (+) da classe Num respeitam a restrição imposta no sistema $\mathcal{O}$, mas a função de leitura da classe Read, e a função fromInt da classe Num não respeitam.

5.2.1.1 Tipos

Define-se o conjunto de tipos deste sistema através da seguinte gramática:

$$
\begin{array}{lcll}
\tau & ::= & \alpha \mid \tau \rightarrow \tau' \mid D\tau_1 \ldots \tau_n & n \geq 0 \\
\sigma & ::= & \tau \mid \forall \alpha.\pi_\alpha \Rightarrow \sigma & \\
\pi_\alpha & = & o_1 : \alpha \rightarrow \tau_1, \ldots, o_n : \alpha \rightarrow \tau_n & n \geq 0,\ o_1, \ldots, o_n \text{ distintos}
\end{array}
$$

Os tipos neste sistema consistem, para além de variáveis de tipo e tipos funcionais, de tipos de dados da forma $D\tau_1 \ldots \tau_n$, sendo n a aridade do construtor de dados D. Um construtor de tipo T é um construtor de tipos de dados D ou o construtor do tipo funcional $(\rightarrow)$, sendo $(\rightarrow)\tau_1\,\tau_2$ sinónimo de $\tau_1 \rightarrow \tau_2$.

Exemplo 5.2.2 *O tipo* Int *é um construtor de aridade 0. Podemos definir um tipo para pares de inteiros da seguinte forma:*

$$\textbf{Par}\ \mathrm{Int}\ \mathrm{Int}$$

Um esquema de tipos é da forma $\forall \alpha.\pi_\alpha \Rightarrow \sigma$, sendo a restrição π_α uma sequência de declarações da forma $o : \alpha \rightarrow \tau$. Quando π_α é uma sequência vazia (), então o esquema de tipos $\forall \alpha.() \Rightarrow \sigma$ corresponde ao esquema de tipos $\forall \alpha.\sigma$ do sistema de Damas-Milner.

Um operador com sobrecarga o, pode aparecer no máximo uma vez numa restrição. As restrições limitam as instâncias de um esquema de tipos, impondo que os operadores com sobrecarga estejam definidos para tipos de dados.

Exemplo 5.2.3 *Consideremos o seguinte esquema de tipos no sistema $\mathcal{O}$:*

$$\sigma = \forall \alpha, \beta.o : \alpha \rightarrow \beta \Rightarrow \alpha \rightarrow int$$

Se temos a declaração de tipo $f : \sigma$, então f é uma função que tem como argumento todos os tipos para os quais está definido o operador com sobrecarga o e como resultado um inteiro.

5.2.1.2 A linguagem de termos do sistema $\mathcal{O}$

No sistema $\mathcal{O}$, não existem declarações de classe. Ao contrário do que acontece no Haskell, não existe uma declaração que agrupe um conjunto de métodos numa mesma classe.

Uma consequência desta eliminação das declarações de classe é a necessidade de declaração de tipo, para cada operador com sobrecarga, nas declarações de instâncias.

A linguagem de termos é a seguinte:

$$M \quad ::= \quad x \mid MM \mid \lambda x.M \mid let\ x = M\ in\ M$$

$$P \quad ::= \quad M \mid \mathbf{inst}\ o : \sigma_T = M\ \mathbf{in}\ P$$

Notação: A letra x é usada para representar variáveis, construtores da linguagem e também operadores com sobrecarga. Usamos variáveis com nome $o, o_1, o_2, \ldots$ para representar operadores com sobrecarga.

Um programa neste sistema é uma sequência de instâncias, seguido de um termo M. Uma declaração de instância é da forma:

$$inst\ o : \sigma_T = M\ in\ P$$

e sobrecarrega o significado do identificador o com a função dada por M em todos os argumentos que são construídos a partir de T.

O esquema de tipos σ_T representa esquemas de tipo fechados que têm T como construtor "mais externo":

$$\begin{array}{lcll} \sigma_T & ::= & T\alpha_1 \ldots \alpha_n \to \tau & fv(\tau) \subseteq \{\alpha_1, \ldots, \alpha_n\} \\ & \mid & \forall \alpha.\pi_\alpha \Rightarrow \sigma'_T & fv(\pi_\alpha) \subseteq fv(\sigma'_T) \end{array}$$

A declaração explícita de σ_T nas declarações de instância é necessária pois a sua ausência poderia permitir declarações em que o construtor de tipo no qual determinado operador com sobrecarga está definido não pode ser determinado de forma única.

Exemplo 5.2.4 *Consideremos a seguinte declaração de instância:*

$$inst\ o = \lambda x.x\ in\ p$$

Sem a declaração de tipo para o, não é possível determinar de forma única o construtor de tipo para o qual estamos a declarar o.

As restrições sintáticas nos esquemas de tipos σ_T garantem três propriedade:

- As instâncias dos operadores com sobrecarga devem funcionar uniformemente para todos os argumentos de um dado construtor de tipos;
- O tipo do argumento deve determinar de forma única o tipo do resultado;
- Todas as restrições devem ser aplicadas a tipos componentes do argumento.

As variáveis com sobrecarga de uma restrição π_α devem ser distintas. A razão de tal imposição deve-se ao facto de um argumento de uma função com sobrecarga determinar de forma única a instância dessa função. Assim, para um tipo argumento τ e um operador com sobrecarga o, existe uma única instância de o com argumento do tipo τ.

Exemplo 5.2.5 *Uma declaração de instância no sistema $\mathcal{O}$:*

$$\boldsymbol{inst}\ (==) : \forall\alpha.((==) : \alpha \to \alpha \to bool) \Rightarrow [\alpha] \to [\alpha] \to bool = listEq$$

em que listEq implementa igualdade entre listas.

Exemplo 5.2.6 *Consideremos o seguinte programa:*

$$
\begin{array}{l}
\textbf{inst}\ eq\ :\ Int\ \to\ Int\ \to\ Bool\ =\ eq\_int\ \textbf{in}\\
\textbf{inst}\ eq\ :\ Float\ \to\ Float\ \to\ Bool\ =\ eq\_float\ \textbf{in}\\
\textbf{inst}\ eq\ :\ Bool\ \to\ Bool\ \to\ Bool\ =\ eq\_bool\ \textbf{in}\\
f\ x\ y\ =\ \textbf{let}\\
\qquad\qquad z\ =\ eq\ 1\ x\\
\qquad\quad \textbf{in}\\
\qquad\quad \textbf{let}\\
\qquad\qquad w\ =\ eq\ 3.0\ y\\
\qquad\quad \textbf{in}\\
\qquad\quad eq\ z\ w
\end{array}
$$

As funções eq_int, eq_float e eq_bool implementam a igualdade para inteiros, vírgula flutuante e booleanos, respectivamente.

5.2.1.3 Regras de Inferência

Seja Γ uma base, P um termo, e σ um esquema de tipos. No sistema $\mathcal{O}$, escreve-se $\Gamma \vdash P : \sigma$ se obtemos $P : \sigma$ pela aplicação das seguintes regras:

$$(Ax) \qquad \Gamma \vdash x : \sigma \quad (x : \sigma \in \Gamma)$$

$$(Set) \qquad \frac{\Gamma \vdash x_1 : \sigma_1 \quad \cdots \quad \Gamma \vdash x_n : \sigma_n}{\Gamma \vdash x_1 : \sigma_1, \ldots, x_n : \sigma_n}$$

$$(\forall I) \qquad \frac{\Gamma, \pi_\alpha \vdash M : \sigma \quad (\alpha \notin tv(\Gamma))}{\Gamma \vdash M : \forall \alpha. \pi_\alpha \Rightarrow \sigma}$$

$$(\forall E) \qquad \frac{\Gamma \vdash M : \forall \alpha. \pi_\alpha \Rightarrow \sigma \quad \Gamma \vdash [\tau/\alpha]\pi_\alpha}{\Gamma \vdash M[\tau/\alpha]\sigma}$$

$$(Abs) \qquad \frac{\Gamma \cup \{u : \tau\} \vdash M : \tau'}{\Gamma \vdash \lambda u.M : \tau \rightarrow \tau'}$$

$$(Apl) \qquad \frac{\Gamma \vdash M : \tau' \rightarrow \tau \quad \Gamma \vdash M' : \tau'}{\Gamma \vdash MM' : \tau}$$

$$(Let) \qquad \frac{\Gamma \vdash M : \sigma \quad \Gamma \cup \{x : \sigma\} \vdash M' : \tau}{\Gamma \vdash \text{let } x = M \text{ in } M' : \tau}$$

$$(Inst) \qquad \frac{\begin{array}{c}(o : \sigma_{T'} \in \Gamma \Rightarrow T \neq T') \\ \Gamma \vdash M : \sigma_T \quad \Gamma \cup \{o : \sigma_T\} \vdash P : \sigma'\end{array}}{\Gamma \vdash \text{inst } o : \sigma_T = M \text{ in } P : \sigma'}$$

O sistema acima é idêntico ao sistema de Damas-Milner excepto por:

- Na regra (∀I), as restrições π_α na variável introduzida α, são passadas do ambiente para o esquema de tipos. A regra (∀E) tem como premissa uma instância da restrição a eliminar. As restrições são derivadas usando a regra (Set).
- Existir uma regra adicional para instâncias ($Inst$) para declaração de instâncias. Nessa regra, a variável com sobrecarga o tem um esquema de tipos explícito σ_T em que o construtor T é diferente para cada instanciação da variável o.

Exemplo 5.2.7 *Consideremos o seguinte programa:*

$$\begin{aligned}&\textbf{inst} + \ : \ int \ \rightarrow \ int \ \rightarrow \ int \ = \ plus\_int \ \textbf{in}\\ &f \ = \ + \, x \ y\end{aligned}$$

Em que plus_int é a soma nos inteiros com tipo $int \rightarrow int \rightarrow int$. *Sendo int o tipo de* x, y *e* $\Gamma = \{x : int, y : int\}$, *temos a seguinte derivação:*

$$\begin{tikzcd}
\Gamma \cup \{+ : int \rightarrow int \rightarrow int\} \vdash + : int \rightarrow int \rightarrow int \arrow[d, "APP"'] & \Gamma \cup \{+ : int \rightarrow int \rightarrow int\} \vdash x : int \arrow[dl, "APP"] \\
\Gamma \cup \{+ : int \rightarrow int \rightarrow int\} \vdash + \ x : int \rightarrow int \arrow[d, "APP"'] & \Gamma \cup \{+ : int \rightarrow int \rightarrow int\} \vdash y : int \arrow[dl, "APP"] \\
\Gamma \cup \{+ : int \rightarrow int \rightarrow int\} \vdash + \ x \ y : int \arrow[d, "INST"'] & \Gamma \vdash plus_int : int \rightarrow int \rightarrow int \arrow[dl, "INST"] \\
\Gamma \vdash \textbf{\textit{inst}} + : int \rightarrow int \rightarrow int = plus_int \ \textbf{\textit{in}} \ + \ x \ y : int &
\end{tikzcd}$$

5.2.2 O HM($\mathcal{O}$)

Apresentamos o sistema $\mathcal{O}$ como instância do HM($\mathcal{X}$). O sistema de restrições que implementamos aqui foi definido em [Sul00] para ilustrar o uso de sistemas de restrições específicos para inferência de tipos, onde foi também demonstrada a equivalência entre HM($\mathcal{O}$) e o sistema $\mathcal{O}$.

5.2.2.1 A linguagem de tipos $\mathcal{T}$

Definimos a linguagem de tipos $\mathcal{T}$ pela seguinte gramática:

$$\tau ::= \alpha \mid \tau \rightarrow \tau' \mid \ T\bar{\tau}$$

onde T representa um construtor de tipo de aridade n. À semelhança do que acontece no sistema $\mathcal{O}$, consideramos os tipos básicos como construtores de tipo de aridade 0.

5.2.2.2 O ambiente inicial Γ_0

Como vimos na secção anterior, as variáveis neste sistema representam operadores com sobrecarga e construtores de dados.

O ambiente inicial neste sistema, contém declarações para as variáveis que representam construtores.

Além disso, para cada operador com sobrecarga o, acrescentamos ao ambiente inicial uma declaração para o da forma:

$$o : \forall \alpha, \beta.(o :: \alpha \rightarrow \beta) \Rightarrow \alpha \rightarrow \beta$$

Estas declarações de tipos, adicionadas ao ambiente inicial, estabelecem as restrições impostas a operadores com sobrecarga no sistema $\mathcal{O}$.

5.2.2.3 O sistema de restrições $\mathcal{O}$

Em HM($\mathcal{O}$) vamos considerar restrições nos esquemas de tipos de duas formas. Restrições da forma $(o :: \alpha \rightarrow \tau)$ restringem as instâncias possíveis da variável de tipo α para um dado operador com sobrecarga o.

As outras restrições são da forma $o :: \sigma_T$, e servem para descrever declarações de instâncias para operadores com sobrecarga. O esquema de tipos σ_T satisfaz as mesmas condições impostas aos esquemas de tipos no sistema $\mathcal{O}$:

$$\begin{array}{rcll} \sigma_T & ::= & T\bar{\alpha} \rightarrow \tau & fv(\tau) \subseteq \bar{\alpha} \\ & | & \forall \alpha.\pi_\alpha \Rightarrow \sigma'_T & fv(\pi_\alpha) \subseteq fv(\sigma'_T) \end{array}$$

onde as restrições em π_α são da forma $(o :: \alpha \rightarrow \tau)$. Assumimos que σ_T é um esquema de tipos fechado.

As regras de normalização para o sistema de restrições $\mathcal{O}$ são as seguintes:

$$over1 \;\; @ \;\; (o :: arrow(\alpha, \tau_1)),\; (o :: arrow(\alpha, \tau_2)) \Longrightarrow unify\_with\_occurs\_check(\tau_1, \tau_2).$$

$$over2 \;\; @ \;\; (o :: arrow(T\bar{\tau_1}, \tau_1)),\; (o :: arrow(T\bar{\tau_2}, \tau_2)) \Longrightarrow unify\_with\_occurs\_check(\bar{\tau_1}, \bar{\tau_2}),$$
$$unify\_with\_occurs\_check(\tau_1, \tau_2).$$

$$over3 \ @ \ (o :: forall(\alpha, D, \tau)), D_1 \Longrightarrow replace(D, [(\alpha, T)], D_1), \\ replace(\tau, [(\alpha, T)], \tau_1) \quad | \ (o :: \tau_1).$$

$$over4 \ @ \ exists(\alpha, \ (o :: arrow(\alpha, \tau))) \Longleftrightarrow true.$$

As regras 1 e 2 garantem que o tipo do argumento determina unicamente o tipo do resultado.

A regra 3 implementa a regra ($\forall$ E), do sistema $\mathcal{O}$. Se para um operador com sobrecarga o temos uma restrição da forma $o :: \forall\alpha.D \Rightarrow \tau$ e temos ainda a restrição $[\tau_1/\alpha]D$, então acrescentamos a restrição $o :: [\tau_1/\alpha]\tau$.

Finalmente, a última regra reflete o facto de termos uma implementação *mundo aberto* do sistema.

O sistema de tipos $\mathcal{O}$ tem uma regra adicional para declarações de instância. Em [Sul00] ao sistema de tipos $HM(\mathcal{O})$ é adicionada uma nova regra para lidar com instâncias. As declarações de instância estão disponíveis antes do processo de inferência. Assim é possível, numa fase de pré-processamento, verificar as declarações de instância e alterar o sistema de restrições.

$$\begin{aligned} pp(\Gamma, \textbf{inst } o : \sigma_T = M \textbf{ in } P) \leftarrow\ & type(C, \Gamma, M, \sigma_T), \\ & add\_ax(o, \sigma_T), \qquad \text{(Inst)} \\ & pp(\Gamma, P). \end{aligned}$$

A função $add\_ax(o, \sigma_T)$ adiciona uma restrição da forma $(o :: \sigma_T)$ ao conjunto de restrições.

Ou seja, cada instância de um operador com sobrecarga, define um axioma da forma $(o :: \sigma_T)$, sendo σ_T a declaração de tipo da instância correspondente de o.

Se, para um dado operador com sobrecarga o, existem duas declarações de instância com o mesmo construtor de tipo T, então temos dois axiomas da forma $(o :: \sigma_T)$ e $(o :: \sigma'_T)$. No entanto, as regras do sistemas de restrições vão impor que σ_T e σ'_T sejam equivalentes.

5.2.2.4 O conjunto de restrições em forma resolvida $\mathcal{S}$

O conjunto $\mathcal{S}$ de restrições em forma resolvida consiste em todas as restrições satisfazíveis da forma:

$$C ::= \text{true} \mid (o :: \alpha \to \tau) \mid C \wedge C \mid \exists\bar{\alpha}.C$$

A projecção em HM($\mathcal{O}$), não é trivial, portanto o conjunto de restrições em forma resolvida contém restrições da forma $\exists\alpha.C$.

5.2.2.5 Exemplos

Exemplo 5.2.8 *Consideremos o seguinte programa:*

$$\begin{aligned} f\ x\ &= \textbf{let} \\ &\qquad g\ y\ =\ eq\ (o\ x)\ y \\ &\quad \textbf{in} \\ &\quad 1 \end{aligned}$$

Os tipos inferidos para f e g, com o ambiente inicial $\Gamma_0 = \{eq : \alpha \to \alpha \to bool, o : \forall\alpha,\beta.(o :: \alpha \to \beta) \Rightarrow \alpha \to \beta, 1 : int\}$, são:

$$\begin{aligned} f &:: \forall\, \alpha,\, \beta\, .(o\ ::\ \alpha\ \to\ \beta)\ \Rightarrow\ \alpha\ \to\ int \\ g &:: \forall\, \beta\, .(o\ ::\ \alpha\ \to\ \beta)\ \Rightarrow\ \beta\ \to\ bool \end{aligned}$$

Exemplo 5.2.9 *Consideremos o seguinte programa:*

$$\begin{aligned} &\textbf{inst} + \ :\ int\ \to\ int\ \to\ int\ =\ plus\_int\ \textbf{in} \\ &\textbf{inst} + \ :\ float\ \to\ float\ \to\ float\ =\ plus\_float\ \textbf{in} \\ &f\ =\ 3\ +\ toInt(2.4\ +\ 2.5) \end{aligned}$$

Onde $plus\_int$ e $plus\_float$ são a soma para inteiros e vírgula flutuante, respectivamente. O tipo inferido para f, com o ambiente inicial $\Gamma_0 = \{toInt : float \to int, + : \forall\alpha,\beta.(+ :: \alpha \to \beta) \Rightarrow \alpha \to \beta, 3 : int, 2.4 : float, 2.5 : float\}$, é:

$$f\ ::\ int$$

Exemplo 5.2.10 *Como último exemplo consideremos o seguinte programa:*

$$\begin{aligned} &\textbf{inst}\ eq\ :\ int\ \to\ int\ \to\ bool\ =\ eq\_int\ \textbf{in} \\ &\textbf{inst}\ eq\ :\ float\ \to\ float\ \to\ bool\ =\ eq\_float\ \textbf{in} \\ &\textbf{inst}\ eq\ :\ bool\ \to\ bool\ \to\ bool\ =\ eq\_bool\ \textbf{in} \end{aligned}$$

$$
\begin{aligned}
f\ x\ y \ = \ & \textbf{let} \\
& \quad z \ = \ eq\ 1\ x \\
& \textbf{in} \\
& \textbf{let} \\
& \quad w \ = \ eq\ 3.0\ y \\
& \textbf{in} \\
& eq\ z\ w
\end{aligned}
$$

Onde eq_int e eq_float e eq_bool são a igualdade para inteiros, vírgula flutuante e booleanos, respectivamente. Os tipos inferidos para z, w e f , com o ambiente inicial $\Gamma_0 = \{eq : \forall\alpha, \beta.(+ :: \alpha \to \beta) \Rightarrow \alpha \to \beta, 1 : int, 3.0 : float\}$*, são:*

$$
\begin{aligned}
& z \ :: \ bool \\
& w \ :: \ bool \\
& f \ :: \ int \ \to \ float \ \to \ bool
\end{aligned}
$$

Capítulo 6

Conclusão

O esquema de inferência de tipos HM($\mathcal{X}$), define sistemas de inferência que são extensões ao sistema de Damas-Milner com restrições.

Nesta dissertação apresentámos uma implementação de um algoritmo geral de inferência de tipos, para sistemas de tipos com restrições, baseada no HM($\mathcal{X}$), usando Prolog e CHR.

Como foi referido na introdução, não nos propusemos a implementar um novo algoritmo de inferência de tipos. O principal objectivo desta dissertação, é verificar quais as vantagens em implementar inferência de tipos para sistemas de tipos com restrições, usando programação por restrições.

A escolha do HM($\mathcal{X}$) como modelo do nosso trabalho vem do facto de existirem já instâncias bem estudadas de sistemas de restrições, para sistemas de tipos interessantes, e de consistir num esquema geral que abstrai propriedades, como a existência de tipos principais e de algoritmos de inferência íntegros e completos.

O CHR, como linguagem de especificação de regras de simplificação de restrições, pareceu-nos uma boa opção, pois permite uma grande flexibilidade na definição dos processos de resolução e dos domínios de restrições.

Analisando o trabalho efectuado, existem alguns pontos que nos parece importante salientar:

- O CHR como linguagem de especificação das regras de normalização do sistema de restrições, permite uma implementação declarativa imediata das regras de simplificação de restrições;

- O uso de módulos CHR para implementar a simplificação de restrições, permite separar o processo de geração de restrições, da sua simplificação/normalização;

- Usar a linguagem Prolog, como linguagem de implementação das regras do sistema de tipos, permite usar a unificação do Prolog para especificar e implementar a unificação entre tipos. Com isso ganhamos uma especificação mais declarativa, face ao uso de linguagens em que a unificação tem que ser implementada.
- As variáveis Prolog propagam as ligações entre as variáveis, dispensando o tratamento explícito das substituições.

6.1 Trabalho futuro

Desta dissertação salientamos dois pontos, como sugestão para trabalho futuro:

1. Propriedades dos programas CHR, tais como a confluência ou a terminação, foram estudadas em [Abd97]. Parece existir uma forte ligação entre estas propriedades e as propriedades comuns dos sistemas de inferência de tipos, tais como a decidibilidade e a existência de tipos principais. Mostrando directamente algumas propriedades dos programas CHR que especificam cada sistema de tipos, poderíamos obter facilmente propriedades dos sistemas de tipos. Isso seria mais uma vantagem em usar CHR na especificação das regras de normalização das restrições sobre os tipos.
2. Aplicação de CHR à especificação de sistemas com subtipos. Trabalho efectuado acerca de sistemas de tipos para linguagem com subtipos [AW93, EST95, Pot96], é baseado em restrições, o que faz com que o CHR pareça ser uma boa ferramenta para estudar e implementar esses sistemas.

6.2 Considerações Finais

Em última análise, pensamos que os objectivos a que nos propunhamos foram cumpridos. Na nossa opinião, o CHR mostrou ser uma boa ferramenta na implementação de módulos de simplificação de restrições, para os sistemas de tipos estudados.

A programação em lógica, dada a existência da unificação como elemento incorporado na linguagem, e o uso da variável lógica, revelou-se um bom paradigma para a especificação e implementação de algoritmos de inferência de tipos baseados em unificação.

Referências

[Abd97] Slim Abdennadher. Operational Semantics and Confluence of Constraint Propagation Rules. In *Principles and Practice of Constraint Programming*, pages 252–266, 1997.

[AF01] Sandra Alves and Mário Florido. Type Inference using Constraint Handling Rules. In *International Workshop on Functional and (Constraint) Logic Programming*, Kiel, Germany, 2001.

[AW93] A. Aiken and E.L. Wimmers. Type inclusion constraints and type inference. In *Conference on Functional Programming Languages and Computer Architechture*, 1993.

[Bar84] Hendrik Pieter Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984.

[Chu33] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33-34:346–366 and 839–864, 1932/33.

[Cur34] H. B. Curry. Funcionality in Combinatory Logic. In *Proc. Nat. Acad. Sci. U.S.A*, volume 20, pages 584–590, 1934.

[DM82] L. Damas and R. Milner. Principal Type-Schemes for Functional Programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico*, pages 207–212, 1982.

[EST95] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to object oriented programming. *Eletronic Notes in Theoretical Computer Science*, 1, 1995.

[Fru98] T. Fruhwirth. Theory and practice of constraint handling rules. *The Journal of Logic Programming*, 37(1–3):95–138, October 1998.

[GSM00] K. Glynn, P. Stuckey, and Sulzmann M. Type Classes and Constraint Handling Rules. In *Workshop on Rule-Based Constraint Reasoning and Programming*, 2000.

[Hin69] J. Roger Hindley. The Principal Type-scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *Conference record of the 14th annual ACM symposium on principles of programming languages*, pages 111 – 119, Munich, Germany, 1987.

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[MS98] K. Marriott and P.J. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.

[NP93] Tobias Nipkow and Christian Prehofer. Type Checking Type Classes. In *Proc. 20th ACM Symp. Principles of Programming Languages*. ACM Press, 1993.

[Oho95] Ohori. A Polymorphic Record Calculus and its Compilation. *ACM TOPLAS*, 6, 1995.

[OWW95] Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *Functional Programming & Computer Architecture*, San Diego, California, June 1995. ACM Press.

[Pot96] François Pottier. Simplifying subtyping constraints. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, volume 31(6), pages 122–133, 1996.

[Rob65] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Sar93] Vijay A. Saraswat. *Concurrent Constraint Programming*. ACM Doctoral Dissertation Awards. MIT Press, 1993.

[SOW99] Martin Sulzmann, Martin Odersky, and Martin Wehr. Type Inference with Constrained Types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[SS94] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. The MIT Press, 1994.

[Sul00] Martin Sulzmann. *A General Framework for Hindley/Milner Type Systems with Constraints*. PhD thesis, Yale University, Department of Computer Science, May 2000.

[Tho96] Simon Thompson. *Haskell: The Craft of Functional Programming.* Addison-Wesley, Harlow, England, 1996.

[Wan87] Mitchell Wand. A Simple Algorithm and Proof for Type Inference. *Fundamenta Infomaticae*, 10:115–121, 1987.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *16'th ACM Symposium on Principles of Programming Languages*, Austin, Texas, January 1989.

Apêndice A

HM($\mathcal{X}$) em Prolog

A.1 Algoritmo de inferência para o HM($\mathcal{X}$) em Prolog

Apresentamos o código Prolog que descreve o algoritmo de inferência de tipos.

```
%------------------------------------------------------------------------
%                    type(C,Gamma,E,Tau)
% C - restrição, Gamma - ambiente inicial, E - expressão, Tau - tipo de E
%------------------------------------------------------------------------
type(C,Gamma,va(X),Tau):-
        member(X,Gamma,forall(Alphas,D,Tau1)),
        new_instance(Alphas,Tau1,Tau,D,D1),
        normalize(D1,C).

type(C,Gamma,lambda(va(X),E),arrow(Alpha,Tau)):-
        type(C,[(X,forall([],[],Alpha))|Gamma],E,Tau).

type(D,Gamma,app(E1,E2),Tau):-
        type(C1,Gamma,E1,Tau1),
        type(C2,Gamma,E2,Tau2),
        union(C1,C2,C),
        unify_with_occurs_check(Tau1,arrow(Tau2,Tau)),
        normalize(C,D).

type(C,Gamma,let(va(X),E1,E2),Tau):-
```

```
        type(C1,Gamma,E1,Tau1),
        gen(C1,Gamma,Tau1,C2,Sigma),
        type(C3,[(X,Sigma)|Gamma],E2,Tau),
        union(C2,C3,D),
        normalize(D,C).

%--------------------------------------------------------------------------
%                    gen(C,Gamma,Tau,C2,Sigma)
% C - restrição, Gamma - ambiente, Tau - tipo, C2 - restrição,
% Sigma - esquema de tipos
%--------------------------------------------------------------------------
gen(C,Gamma,Tau,C2,forall(Vars,C1,Tau)):-
        fvlist(Gamma,[],L-[]),
        fv(Tau,L,L1-[]),
        fvlist(C,L,L2-[]),
        union(L1,L2,Vars),
        split(C,Vars,D,C1),
        newconstr(D,C1,Vars,C2).

%--------------------------------------------------------------------------
%                    normalize(D,C)
% D - restrição a normalizar, C - restrição normalizada
%--------------------------------------------------------------------------
normalize([],C):- !, findall_constraints(_,CId), removeId(CId,C).
normalize([X|R1],C):- call(X), normalize(R1,C).

%--------------------------------------------------------------------------
%                    removeId(CId,D)
% CId - lista de restrições da forma C # Id
% D - lista de restrições sem Id
%--------------------------------------------------------------------------
removeId([],[]).
removeId([C # Id|RC],[C|R]):- remove_constraint(Id), removeId(RC,R).

%--------------------------------------------------------------------------
%                    split(C,Vars,D,C')
% C - restrição, Vars - conjunto de variáveis, D - restrição sem
```

```
% variáveis em Vars, C' - restrição com variáveis em Vars
%------------------------------------------------------------------------------
split([],_,[],[]).
split(C,[],C,[]).
split([D|RC],Vars,[D|RD],C1):-
        fv(D,[],L-[]),
        intersect(L,Vars,[]),!,
        split(RC,Vars,RD,C1).
split([C|RC],Vars,D,[C|RC1]):-
        split(RC,Vars,D,RC1).

%------------------------------------------------------------------------------
%                    newconstr(D,C,Vars,C')
% D e C - restrições, Vars - variáveis, C'- restrição da forma D /\
% exists Vars.C
%------------------------------------------------------------------------------
newconstr(D,C,[],D).
newconstr(D,[],_,[]).
newconstr(D,C,Vars,[exists(Vars,C)|D]).

%------------------------------------------------------------------------------
%                    new_instance(Vars,Tau,Tau',D,D')
% Vars - variáveis, Tau - tipo, Tau'- tipo da forma [Betas/Vars] Tau,
% D - restrição,  D'- restrição da forma [Betas/Vars] D
%------------------------------------------------------------------------------
new_instance(Alphas,Tau1,Tau,D1,D):-
        newvars(Alphas,Betas),
        replace(Tau1,Betas,Tau),
        replacelist(D1,Betas,D).

%------------------------------------------------------------------------------
%                    replace(T1,Vars,T2)
% T1 - termo ou restriçao, Vars - lista de pares de variáveis (alpha,beta)
% T2 - termo ou restrição com as variáveis substituídas
%------------------------------------------------------------------------------
replace(Z,[],Z):- !.
replace(X,Betas,Y):- var(X), member2(X,Betas,Y),!.
```

```
replace(X,Betas,X):- var(X), !.
replace(X,Betas,X):- atom(X), !.
replace(arrow(A1,B1),Betas,arrow(A2,B2)):-
        replace(A1,Betas,A2),
        replace(B1,Betas,B2),!.
replace(Alpha kind K,Betas,BAlpha kind BK):- replace(Alpha,Betas,BAlpha),
        replace(K,Betas,BK).
replace(labtype(L,Tau),Betas,labtype(L,BTau)):- replace(Tau,Betas,BTau).
replace((L,Tau),Betas,(L,BTau)):- replace(Tau,Betas,BTau).
replace(rectype(L),Betas,rectype(BL)):- replacelist(L,Betas,BL).
replace(class(A1,C),Betas,class(A2,C)):- replace(A1,Betas,A2).
replace(over(O,T1),Betas,over(O,T2)):- replace(T1,Betas,T2).
replace(exists(Vars,Const),Betas,exists(BVars,BConst)):-
        replacelist(Vars,Betas,BVars),
        replacelist(Const,Betas,BConst).
replace(forall(Vars,Const,Sigma),Betas,forall(BVars,BConst,BSigma)):-
        replacelist(Vars,Betas,BVars),
        replacelist(Const,Betas,BConst),
        replace(Sigma,Betas,BSigma).

%---------------------------------------------------------------------------
%                    replacelist(L1,Vars,L2)
% L1 - lista de termos ou restrições, Vars - lista de pares de variáveis
% L2 - lista de termos ou restrições com as variáveis substituídas
%---------------------------------------------------------------------------
replacelist([],_,[]):- !.
replacelist([X|R],Betas,[XBetas|RBetas]):- replace(X,Betas,XBetas),
        replacelist(R,Betas,RBetas).

%---------------------------------------------------------------------------
%                    fv(T,Vars,V)
% T - tipo ou restrição, Vars - conjunto de variáveis
% V - variáveis livres de T que não pertencem a Vars
%---------------------------------------------------------------------------
fv(X,Vars,L-L):- atom(X), !.
fv(X,Vars,L-L):- var(X), member(X,Vars),!.
fv(X,Vars,[X|L]-L):- var(X), !.
```

```
fv(arrow(A,B),Vars,L1-L3):- !, fv(A,Vars,L1-L2),  fv(B,Vars,L2-L3).
fv(rectype(L),Vars,L1-L2):- !, fvlist(L,Vars,L1-L2).
fv(labtype(L,Labels),Vars,L1-L2):- !, fv(Labels,Vars,L1-L2).
fv(T kind K,Vars,L1-L3):- fv(T,Vars,L1-L2), fv(K,Vars,L2-L3).
fv(class(Alpha,C),Vars,L1-L2):- fv(Alpha,Vars,L1-L2).
fv(over(O,Tau),Vars,L1-L2):- fv(Tau,Vars,L1-L2).
fv(forall(V,_,Tau),Vars,L1-L2):- append(V,Vars,VVars), fv(Tau,VVars,L1-L2).
fv(exists(V,C),Vars,L1-L2):- append(V,Vars,VVars), fvlist(C,VVars,L1-L2).

%----------------------------------------------------------------------------
%                     fvlist(L,Vars,LV)
% L - lista de tipos ou restrições, Vars - conjunto de variáveis
% LV - variáveis livres de L que não pertencem a Vars
%----------------------------------------------------------------------------
fvlist(L1,_,L-L):- var(L1), !.
fvlist([],_,L-L):- !.
fvlist([(X,V)|R],Vars,L1-L3):- !, fv(V,Vars,L1-L2),
        fvlist(R,Vars,L2-L3).
fvlist([V|R],Vars,L1-L3):- fv(V,Vars,L1-L2), fvlist(R,Vars,L2-L3).
```

A.2 Módulos em CHR

A.2.1 Sistema com estruturas com campos

Módulo em CHR de normalização de restrições para o sistema com estruturas de Ohori descrito no capítulo 4.

```
records1   @ rectype(R) kind labtype(L,Taui) <=> member(L,R,Taui) | true.
records2   @ Tau kind labtype(L,Tau1) , Tau kind labtype(L,Tau2) ==>
             Tau1 = Tau2.
records3   @ rectype(R) kind labtype(L,Tau2) ==> member(L,R,Tau1) |
                                                 Tau2 = Tau1.
records4   @ exists([Alpha],K)  <=> kinds(Alpha,K), fvlist(K,V),
                                    notin(Alpha,V) | true.
```

A.2.2 Sistema com classes de tipos

Módulo em CHR de normalização de restrições para o sistema de Nipkow e Prehofer descrito na secção 5.1.

```
class1  @ class(Alpha,[C]) <=> class(Alpha,C).
class2  @ class(Alpha,[C_1|RC]) <=> class(Alpha,C_1), class(Alpha,RC).
class3  @ exists([Alpha],S) <=> classes(Alpha,S) | true.
```

A.2.3 Sistema $\mathcal{O}$

Módulo em CHR de normalização de restrições para o sistema $\mathcal{O}$ descrito na secção 5.2.

```
over1  @ over(O,arrow(Alpha,Tau1)), over(O,arrow(Alpha,Tau2)) ==>
         unify_with_occurs_check(Tau1, Tau2).
over2  @ over(O,arrow(Tc1,Tau1)), over(O,arrow(Tc2,Tau2)) ==>
         Tc1 =.. [T|Alphas1], Tc2 =.. [T|Alphas2] |
         unify_with_occurs_check(Alphas1, Alphas2),
         unify_with_occurs_check(Tau1, Tau2).
over3  @ over(O,forall([Alpha],[D],Tau)), D1 ==> replace(D,[(Alpha,T)],D1),
         replace(Tau,[(Alpha,T)],Tau1) | over(O,Tau1).
over4  @ exists([Alpha],[over(O,arrow(Alpha,Tau))]) <=> true.
```