CLASSES DE TIPOS OPCIONAIS E COM VÁRIOS PARÂMETROS EM HASKELL

RODRIGO GERALDO RIBEIRO

CLASSES DE TIPOS OPCIONAIS E COM VÁRIOS PARÂMETROS EM HASKELL

Projeto de tese apresentado ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

ORIENTADOR: CARLOS CAMARÃO DE FIGUEIREDO

Belo Horizonte - MG

Agosto de 2010

"Você acha que uma vida como essa, com tal objetivo, seria árdua demais, despida de coisas agradáveis? Então não aprendeu ainda que não há mel mais doce que o do conhecimento."

(Friedrich Nietzsche - Filósofo Alemão)

Resumo

A introdução de classes de tipos com múltiplos parâmetros em Haskell tem sido dificultada devido a problemas associados a ambiguidade, que ocorre devido a uma falta de especialização durante a inferência de tipos. Este trabalho propõe um novo sistema de tipos para Haskell que permite a definição de classes com múltiplos parâmetros sem a necessidade de extensões como dependências funcionais ou famílias de tipos e a declaração de símbolos sobrecarregados sem a necessidade prévia de definir uma classe de tipos para estes símbolos. Além disso, um algoritmo de inferência de tipos correto e completo com respeito ao sistema de tipos proposto é implementado como parte deste trabalho.

Abstract

The introduction of multi-parameter type classes in Haskell has been hindered because of problems associated to ambiguity, which occur due to the lack of type specialization during type inference. This work proposes a new type system for Haskell that supports the definition of multi-parameter type classes without the need of any extensions like functional dependencies or type families and allows the definition of overloaded symbols without the corresponding type class. A type inference algorithm sound and complete with respect to the proposed type system is presented and implemented.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Linguagens de programação modernas têm evoluído no sentido de utilizar sistemas de tipos mais flexíveis, que permitem aos programadores escrever programas sem restrições, como se estivessem programando em linguagens não tipadas ou com tipagem dinâmica, mas garantindo que erros de tipo não ocorram durante a execução do programa. O uso de sistemas de inferência de tipos, em vez de sistemas de verificação de tipos, é um exemplo de uma característica importante nessa direção, que, contudo, ainda introduz restrições nas linguagens devido ao desejo ou necessidade de manter o processo de inferência de tipos decidível e eficiente.

A maioria das linguagens atuais possui sistemas de tipos com suporte a definições polimórficas que permitem a definição de funções que operam sobre valores de diferentes tipos. Dá-se o nome de *polimorfismo universal*, *polimorfismo paramétrico*, *polimorfismo via-let*[1] ou *polimorfismo de Damas-Milner* [Mitchell, 1996] ao mecanismo que permite a definição de funções que comportam-se de maneira idêntica sobre todos os valores de tipos que são instâncias de um determinado tipo principal [Mitchell, 1996]. Diversas funções presentes em bibliotecas para manipulação de estruturas de dados são exemplos de funções que possuem este tipo de polimorfismo: contar o número de elementos de uma determinada estrutura de dados, selecionar (filtrar) um subconjunto de elementos de uma estrutura de dados, aplicar uma função a cada um dos elementos de uma determinada estrutura de dados, são alguns dos muitos exemplos de funções que utilizam polimorfismo paramétrico.

O tipo principal de expressões (e portanto de definições de funções) é em geral inferido automaticamente pelo compilador (ou interpretador) da linguagem, de acordo com tipos de constantes e funções predefinidas.

Porém, muitas vezes, deseja-se definir funções que não operam da mesma maneira

[1] Em inglês: *let-polymorphism*

sobre valores de qualquer tipo que é instância de um tipo principal, mas sim funções que possuem comportamento diferente de acordo com o tipo do valor para o qual estas são aplicadas. Dá-se o nome de *polimorfismo ad-hoc* ou *polimorfismo de sobrecarga* ao mecanismo presente em linguagens que permitem definições de funções que comportam-se desta maneira. Exemplos destas funções incluem: teste de igualdade, comparação referente a ordem de valores (menor-que, maior-que), *parsers*, *pretty-printers*, etc.

Linguagens de programação como *Java* e *C++* permitem definições que utilizam *polimorfismo paramétrico* e uma forma restrita de *sobrecarga* denominada *sobrecarga independente de contexto* [Watt, 1990], onde a resolução de qual função sobrecarregada será utilizada é feita com base apenas nos tipos dos argumentos fornecidos em uma chamada de função. Uma política de sobrecarga independente de contexto simplifica a resolução de sobrecarga e a detecção de ambiguidades, mas é restritiva. Por exemplo, constantes não podem ser sobrecarregadas e não é permitida a sobrecarga de funções onde apenas o tipo do valor retornado é diferente para as várias definições. Isso ocorre, por exemplo, no caso de uma função de leitura ou conversão de valores para *strings*, como a função *read* definida na biblioteca padrão de Haskell [Jones, 2002]. Esta função é sobrecarregada em Haskell para diversos tipos básicos da biblioteca padrão (*Int*, *Float*, *Bool*, entre outros). Cada uma destas definições tem um tipo que é uma instância do tipo polimórfico $\forall\alpha.String \rightarrow \alpha$. Um sistema de tipos que adote uma política dependente do contexto permite a resolução de sobrecarga em declarações como: $\lambda x.\, read\ x == \text{“}abc\text{”}$. Neste exemplo, o tipo de *read* é determinado como sendo $String \rightarrow String$.

A linguagem *Haskell* permite combinar o *polimorfismo paramétrico* com o suporte à sobrecarga. Símbolos sobrecarregados podem ser definidos mediante a declaração de *Classes de Tipos* [Wadler & Blott, 1989]. Cada declaração de classe define o nome da classe, um ou mais parâmetros (definidos como variáveis de tipos) e nomes ou símbolos, junto com seus respectivos tipos principais, sendo que as variáveis de tipos usadas nesses tipos principais devem ser instâncias de cada um dos parâmetros da classe. Implementações de símbolos sobrecarregados são feitas em declarações de instâncias. Uma declaração de instância são definidas as funções para os nomes especificados em uma classe, com tipos que devem ser instâncias do tipo principal especificado na classe.

Na atual definição da linguagem [Jones, 2002], são permitidas classes com, no máximo, um parâmetro. Classes com mais de um parâmetro não foram introduzidas na definição da linguagem devido a dificuldades existentes no tratamento de tipos ambíguos [2] que podem surgir no uso de símbolos sobrecarregados. Os atuais compiladores

[2] Uma expressão e é considerada ambígua se seu tipo pode ser produzido por duas ou mais derivações de tipos e estas atribuem diferentes denotações para e [Mitchell, 1996].

/ interpretadores de Haskell permitem a utilização de classes com múltiplos parâmetros utilizando extensões do sistema de tipos da linguagem. Uma destas extensões utiliza as chamadas *dependências funcionais* [Jones, 2000]. Uma dependência funcional permite ao programador especificar que um dos parâmetros da classe deve ser unicamente determinado por um ou mais parâmetros da classe. Apesar de úteis, em algumas situações dependências funcionais não podem ser utilizadas, uma vez que pode não existir uma dependência funcional entre os parâmetros de uma classe.

O dilema atual enfrentado pelos projetistas de Haskell é que classes com múltiplos parâmetros são muito úteis e devem ser introduzidas na linguagem, mas as extensões propostas para solucionar os problemas de ambiguidade devido a utilização destas não resolvem completamente o problema e adicionam uma complexidade extra ao sistema de tipos de Haskell.

1.1 Objetivos

O objetivo principal deste trabalho é a elaboração de um novo sistema e algoritmo de inferência de tipos para Haskell que dê suporte a classes de tipos com múltiplos parâmetros sem a necessidade de extensões como dependências funcionais e que permita a definição de símbolos sobrecarregados sem a necessidade prévia de declarar uma classe de tipos. Além da definição do sistema e do algoritmo de inferência de tipos, o presente trabalho pretende: a implementação de um *front-end* de um compilador Haskell que implemente o algoritmo de inferência proposto e a demonstração de propriedades de correção e tipagem principal do algoritmo em relação ao sistema de tipos [Mitchell, 1996, Wells, 2002, Jim, 1996].

1.2 Contribuições

1. Definição e formalização de um novo algoritmo de inferência e um novo sistema de tipos para Haskell, que dê suporte a definição e uso de classes de tipos com múltiplos parâmetros que seja correto, possua a propriedade de tipagem principal e que permita a definição opcional de classes de tipos se o programador julgar adequado.

2. Implementação de um protótipo de um *front-end* para Haskell que implemente o algoritmo de inferência proposto e permita sua utilização para a verificação e inferência de tipos de bibliotecas Haskell que até o presente momento são desenvolvidas utilizando-se alguma extensão para suporte a classes de ti-

pos com múltiplos parâmetros implementada em compiladores como o GHC [S. P. Jones and others, 1998].

1.3 Metodologia

A definição, formalização e implementação do sistema de tipos proposto envolve as seguintes etapas:

1. Definição de um sistema e de um algoritmo de inferência de tipos para permitir classes com múltiplos parâmetros em Haskell e implementação de um protótipo baseado nesse algoritmo.

2. Definição de um sistema e de um algoritmo de inferência de tipos que permita a declaração opcional de classes de tipos, considerando conhecido o conjunto de todas as instâncias dessa classe. Implementação de um protótipo que baseado neste algoritmo.

3. Demonstração das propriedades de tipo e tipagem principal dos algoritmos em relação aos sistemas de tipos propostos.

4. Prova de terminação dos algoritmos de inferência de tipos definidos.

1.4 Organização do Trabalho

Além deste capítulo introdutório, este trabalho é dividido em três partes. A primeira delas compreende os capítulos 2 e 3 que apresentam a linguagem Haskell e sua abordagem para polimorfismo de sobrecarga. A segunda parte compreende o capítulo 4 que apresenta a definição formal do sistema de tipos elaborado, suas propriedades e descreve a implementação do *front-end* que o utiliza. A terceira e última parte é formada pelo capítulo 5 que apresenta o sumário da tese e o cronograma das atividades que ainda serão desenvolvidas.

Capítulo 2

A Linguagem Haskell

Este capítulo apresenta uma breve introdução à linguagem Haskell. Leitores familiarizados com esta linguagem podem continuar a leitura a partir do Capítulo 4.

"Haskell é uma linguagem de propósito geral, puramente funcional, que incorpora muitas inovações recentes em seu projeto. Haskell provê funções de alta ordem, semântica não-estrita, sistema de tipos polimórfico com inferência e verificação estática, tipos de dados algébricos definidos pelo usuário, casamento de padrões, sintaxe especial para listas, um sistema de módulos, um sistema de E/S monádico e um rico conjunto de tipos de dados primitivos, incluindo listas, arranjos, inteiros de precisão fixa e arbitrária e números de ponto flutuante. Haskell é o ápice da solidificação de vários anos de pesquisa em linguagens funcionais não-estritas" (Definição da Linguagem Haskell [Jones, 2002]).

Para a apresentação de diversas características da linguagem, considere o trecho de programa mostrado na Figura 2.1.

2.1 Módulos

Programas em Haskell são compostos por um conjunto de *módulos*. Módulos provêem uma forma para o programador re-utilizar código e controlar o espaço de nomes em programas. Cada módulo é composto por um conjunto de *declarações*, que podem ser: declarações de classes, instâncias, tipos de dados, sinônimos de tipos, valores e funções. A Figura 2.1 mostra um trecho de código de um módulo chamado `Table` que implementa operações em uma tabela representada por uma lista de pares chave-valor. Este módulo define a constante `empty` e as funções `insert`, `member`, `search`, `remove` e `update` para manipulação de tabelas.

```
module Table where


type Table a = [(String, a)]

empty :: Table a
empty = []


insert ::  String → a → Table a → Table a

insert s a t
         | member s t = t
         | otherwise = (s, a) : t


member ::  String → Table a → Bool
member s t = not $ null [p | p ← t, fst p /= s]


search ::  String → Table a → a
search s t = snd (head [p | p ← t, fst p == s])


update ::  String → a → Table a → Table a

update s a [] = error "Item not found!"
update s a (x:xs)
              | s == (fst x) = (s, a) : xs
              | otherwise = update s a xs


remove ::  String → Table a → (a, Table a)

remove s [] = error "Item not found!"
remove s (x:xs)
            | s == (fst x) = (snd x, xs)
            | otherwise = remove s xs
```

Figura 2.1. Um Módulo em Haskell

2.2 Anotações de Tipo

No módulo `Table`, cada definição é precedida por uma correspondente *anotação de tipo*.

Todos os símbolos definidos no módulo `Table` são *polimórficos*. Por exemplo, a constante `emptyTable`, possui o tipo `Table a`, que corresponde a um sinônimo para o tipo `[(String, a)]`, que indica que este símbolo pode assumir diferentes tipos de acordo com o valor da *variável de tipo* `a`; caso `a` tenha o valor `Int`, este tipo será o tipo monomórfico `[(String, Int)]`; se `a` tiver o valor `Bool`, então este tipo será o tipo monomórfico `[(String, Bool)]`; se `a` tiver o valor `[b]`, este será o tipo polimórfico `[(String, [b])]` etc.

Tipos funcionais especificam os tipos do parâmetro e do resultado de uma função (os quais podem também ser tipos funcionais). O símbolo `search` possui a seguinte anotação de tipo: `String` $\rightarrow$ `Table a` $\rightarrow$ `a`, que especifica que esta função recebe como parâmetro um valor do tipo `String` e uma lista de pares compostos por uma `String` e um elemento de um tipo qualquer e retorna como resultado um elemento deste tipo. Em geral é como dizer, informalmente, que `search` recebe dois parâmetros (um de cada "vez"), um valor de tipo `String` e uma lista de pares.

Cabe ressaltar que, com poucas exceções, anotações de tipos são opcionais em programas Haskell, uma vez que o compilador é capaz de inferir o tipo principal para cada expressão. Este processo de determinar o tipo principal para uma expressões é chamado de *inferência de tipo*. Caso o programador forneça uma anotação de tipo para uma expressão, o compilador verifica se a anotação especificada pode ter o tipo anotado. Este processo de verificação é chamado de *verificação de tipo*.

2.3 Sintaxe de Listas

Listas são estruturas de dados usadas comumente para modelar diversos problemas. Por isto, existe em Haskell uma sintaxe especial para representar este tipo de dados. O tipo de dados `[a]` pode ser definido indutivamente como a união disjunta de uma lista vazia, representada por `[]`, com o conjunto de valores `x : xs` contendo um primeiro elemento `x` de tipo `a`, seguido de uma lista `xs`. Os símbolos `[]` e `:` são *construtores de valores* do tipo lista, cujos tipos são respectivamente `[a]` e `a` $\rightarrow$ `[a]` $\rightarrow$ `[a]`. O uso de `[a]` (em vez de `List a`) é uma primeira forma de sintaxe especial para (tipos de) listas. O uso dos construtores `[]` e `(:)`, sendo o segundo usado de forma infixada, é outra notação especial para a construção de listas.

Uma outra forma de sintaxe especial para listas é mostrada a seguir:

```
l = [True, False]
```

corresponde a uma abreviação para

```
l = True : (False : []).
```

No módulo `Table`, a função `member` usa outro tipo de sintaxe especial para listas, que é baseada em notação comumente usada para definição de conjuntos. Esta função poderia ser definida usando notação de conjuntos como:

member s t = $\{ p \mid p \in t \wedge (\text{fst } p) = s\} \neq \emptyset$

O último tipo de *açúcar sintático* disponível na linguagem Haskell para listas é utilizado para facilitar a definição de seqüências aritméticas:

- `['a'..'z']` : lista de todas as letras minúsculas do alfabeto.
- `[0, 2..]`: lista de números naturais pares.
- `[0..]`: lista de todos os números naturais.

2.4 Casamento de Padrões

O *casamento de padrões* desempenha um papel fundamental nas definições de funções em linguagens funcionais modernas, por meio de equações. A função `remove`, definida no módulo `Table`, é um exemplo de definição que utiliza casamento de padrão sobre listas. A definição desta função é composta por duas equações alternativas, cada uma especificando o resultado correspondente ao padrão da lista recebida como argumento: a primeira equação o padrão `[]`, e a segunda equação utiliza o padrão `(x:xs)`.

Guardas

A definição da função `insert` é um exemplo de definição que utiliza *definições com guardas*, que permitem a definição de alternativas para uma mesma equação. A alternativa a ser executada é a primeira, na ordem textual, para qual a guarda (expressão booleana) especificada na definição resulta valor verdadeiro.

2.5 Tipos de Dados Algébricos

A seguir, nas Figuras 2.2 e 2.3 são mostradas declarações de um tipo de dados algébrico e de uma função que recebe valores deste tipo como argumento, com o objetivo de

ilustrar características básicas da definição e uso de valores de tipos de dados algébricos em Haskell.

```
data Maybe a = Nothing | Just a
mapMaybe ::  (a → b) → Maybe a → Maybe b
mapMaybe f (Just x) = Just (f x)
mapMaybe f Nothing = Nothing
```

Figura 2.2. Definição de um tipo de dados algébrico e uma função que o utiliza.

A primeira linha ilustra a definição de um tipo algébrico: apalavra reservada `data` declara `Maybe` como sendo um novo *construtor de tipos* que possui dois *construtores de dados*: `Nothing` e `Just`. O tipo `Maybe a` é polimórfico, ou seja, quantificado universalmente sobre a variável de tipo `a`: para cada tipo `t` atribuído à variável de tipo `a`, o construtor de tipos `Maybe` define um novo tipo de dados, `Maybe t`. Os valores de um tipo `Maybe t` podem ter duas formas: `Nothing` ou `(Just x)`, onde `x` corresponde a um valor do tipo `t`. Construtores de dados podem ser utilizados em padrões para decompor valores do tipo `Maybe` ou em expressões para construir valores deste tipo. Ambos os casos estão ilustrados na definição de `mapMaybe`.

Tipos de dados algébricos em Haskell constituem uma *soma de produtos*. A definição do tipo de dados `Tree a` é uma folha (`Leaf`) que corresponde a um produto trivial contendo somente um tipo, e um nodo (`Node`), que corresponde a um produto contendo um elemento do tipo `a` e dois elementos de tipo `Tree a` (sub-árvores esquerda e direita). O tipo de dados `Tree a` corresponde á soma de dois produtos, um correspondente ao construtor de dados `Leaf` e outro ao construtor `Node`.

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
```

Figura 2.3. Tipo de dados algébrico.

2.6 Conclusão

Este capítulo apresentou uma introdução à linguagem Haskell e suas principais características: sistema de módulos, anotações de tipos, açúcar sintático para listas, casamento de padrões e definições de tipos de dados algébricos. Para cada uma destas características foram apresentados exemplos ilustrativos de sua utilização. O próximo capítulo abordará outro recurso importante de Haskell — a possibilidade de definir símbolos sobrecarregados.

Capítulo 3

Polimorfismo de Sobrecarga em Haskell

Este capítulo apresenta uma breve introdução à abordagem adotada na linguagem Haskell para sobrecarga. Leitores familiarizados com estes conceitos podem continuar a leitura a partir do Capítulo 4.

3.1 Introdução

Strachey, em 1964, foi o primeiro a utilizar o termo *polimorfismo ad-hoc* para se referir a funções polimórficas que podem ser aplicadas a argumentos de diferentes tipos, mas que comportam-se de acordo com o tipo do argumento para as quais são aplicadas [Strachey, 2000]. Neste texto será usado, preferencialmente, o termo *polimorfismo de sobrecarga* para denominar este tipo de função. Linguagens que provêem suporte ao polimorfismo de sobrecarga permitem ao programador fazer várias definições, todas com o mesmo nome. A tarefa de determinar qual função é chamada pode ser realizada estaticamente pelo compilador, que toma esta decisão com base em informações do contexto onde o nome da função é usado, ou pode ser realizada dinamicamente, ou seja, durante a execução do programa.

Ao contrário do *polimorfismo paramétrico*, a importância do polimorfismo de sobrecarga é muitas vezes subestimada, considerando que este não aumenta a expressividade de uma linguagem, pois poderia ser eliminado por uma renomeação adequada de símbolos. Todavia, a importância do polimorfismo de sobrecarga não está em evitar a poluição do espaço de nomes, mas na propriedade de que expressões e nomes definidos utilizando símbolos sobrecarregados podem ser usados em contextos que podem

requerer valores de tipos distintos [Camarao & Figueiredo, 1999a].

Linguagens que provêem polimorfismo de sobrecarga utilizam uma *política de sobrecarga* (*overloading policy*) que visa extender a possibilidade de sobrecarga de modo a permitir que um maior número de programas que utilizam símbolos sobrecarregados sejam considerados corretos e, ao mesmo tempo, estabelece regras que limitam a sobrecarga, para permitir que o processo de inferência de tipos seja eficiente. Uma estratégia de sobrecarga pode ser caracterizada como *dependente de contexto* ou *independente de contexto* [Watt, 1990]. Em uma estratégia de sobrecarga independente de contexto, se f é um símbolo sobrecarregado então, para cada aplicação $f\ e$, a decisão sobre qual função f será aplicada é determinada de acordo com o tipo da expressão e. Por sua vez, uma estratégia de sobrecarga dependente de contexto pode utilizar o contexto no qual a expressão $f\ e$ é usada para determinar qual definição do símbolo f será usada.

Estratégias independentes de contexto para o polimorfismo de sobrecarga são utilizadas em diversas linguagens populares, como `C++` e `Java`, para métodos definidos em uma mesma classe (desconsiderando o fato de que o mecanismo de associação dinâmica em chamadas de métodos − no qual o método a ser chamado é determinado de acordo com o tipo do objeto usado (como alvo) na chamada de método − pode ser visto como uma forma de resolução de sobrecarga). Apesar da abordagem de sobrecarga independente de contexto permitir soluções simples para a resolução da sobrecarga, ela é muito restritiva. Por exemplo, símbolos como *read*, cujas definições possuem tipos que são instâncias de $\forall a.String \rightarrow a$, não podem ser sobrecarregados, uma vez que não é possível determinar, utilizando apenas o tipo da expressão fornecida como argumento para *read* para qual tipo deverá ser instanciada a variável a. Uma estratégia de sobrecarga dependente de contexto, por outro lado, permite tais definições; por exemplo, o tipo de *read* em $\lambda x.\ read\ x == \text{“}a\ string\text{”}$ pode ser inferido como $String \rightarrow String$.

Muitos sistemas de tipo que provêem suporte a polimorfismo de sobrecarga têm adotado uma estratégia dependente de contexto para sobrecarga, por esta ser menos restritiva. Nesta classe de sistemas de tipos estão incluídos o sistema *CT* [Camarao & Figueiredo, 1999a] e o sistema de classes tipos utilizado pela linguagem *Haskell* [Jones, 2002, Wadler & Blott, 1989].

Desde sua proposta original em [Wadler & Blott, 1989], o sistema de classes de tipos sofreu mudanças para a inclusão de diversas extensões. Em sua maioria, estas extensões tinham o intuito de permitir a utilização de classes de tipos com múltiplos parâmetros[1]. Dentre estas podemos ci-

[1] do inglês: *Multi-Parameter Type Classes*.

tar: Classes de Tipos Paramétricas [Chen et al., 1992], Dependências Funcionais [Jones, 2000, Jones & Diatchki, 2009, Sulzmann et al., 2006a] e Famílias de tipos[2] [Schrijvers et al., 2008, Chakravarty et al., 2005b].

3.2 Polimorfismo de Sobrecarga em Haskell

Esta seção descreve o polimorfismo de sobrecarga em Haskell, que é baseado em classes de tipos.

3.2.1 Classes de tipo

Classes de tipo em Haskell [Jones, 2002, Hudak et al., 2007] permitem ao programador definir símbolos sobrecarregados e seus respectivos tipos, que podem ser instanciados então para diferentes tipos, definidos como instâncias de classes.

Uma declaração de instância de uma determinada classe fornece a definição para os símbolos desta classe, para tipos específicos para cada parâmetro da classe. Como um primeiro exemplo (baseado em um exemplo de [Hudak et al., 2007]), considere a classe `Eq`, que possui um único parâmetro, e duas instâncias definidas para `Int` e `Bool`, apresentadas na figura 3.1.

```
class Eq a where
   (==), (/=) :: a → a → Bool
   x == y = not (x /= y)
   x /= y = not (x == y)

instance Eq Int where
   x == y = primEqInt x y

instance Eq Bool where
   True == True = True
   False == False = True
   _ == _ = False

instance Eq a ⇒ Eq [a] where
   [] == [] = True
   (x:xs) == (y:ys) = (x == y) && (xs == ys)
   xs /= ys = not (xs == ys)
```

Figura 3.1. Exemplo de classe de tipos e instâncias

[2] do inglês: *Type families*.

Suponha que `primEqInt` seja uma função primitiva, de tipo `Int` → `Int` → `Bool`, que verifique a igualdade de dois números inteiros. Considerando as duas instâncias definidas para a classe `Eq`, tem-se que as seguintes expressões `2 == 3` e `False /= False` são bem tipadas. De maneira similar, a seguinte declaração polimórfica também é bem tipada:

```
member x [] = False
member x (y:ys) = (x == y) || (member x ys)
```

Figura 3.2. Função polimórfica cujo tipo é restringido pela classe `Eq`.

A função `member`, definida na Figura 3.2, tem um tipo que pode ser denotado por `member :: Eq a ⇒ a → [a] → Bool`, sendo que `Eq a` é uma restrição sobre o tipo polimórfico `a → [a] → Bool`, que limita os tipos para os quais a variável `a` pode ser instanciada aos tipos pertencentes à classe `Eq`.

Classes de tipos podem ser declaradas de maneira a formar hierarquias. Por exemplo:

```
class Eq a where
   (==), (/=) :: a  →  a  → Bool
class Eq a ⇒  Ord a where
   (>), (<) :: a  →  a  → Bool
```

Figura 3.3. Exemplo de hierarquia de classes de tipos.

Na Figura 3.3, a classe `Ord` é definida como *subclasse* de `Eq`. Assim sendo, um tipo somente pertencerá a classe `Ord` se este já pertencer a classe `Eq`. A formação de hierarquia de classes pode simplificar os tipos de expressões envolvendo símbolos sobrecarregados, como no seguinte exemplo:

```
search y [] = False
search y (x:xs) = if x == y then True
                  else if x < y then False else search y xs
```

O tipo inferido para esta função é: `search ::  Ord a ⇒ a → [a] → Bool`. Caso a classe `Ord` não fosse definida como subclasse de `Eq` teríamos:

```
search ::  (Ord a, Eq a) ⇒ a → [a] → Bool.
```

Uma interessante característica de funções sobrecarregadas definidas por classes de tipo é que estas podem ser traduzidas em funções não sobrecarregadas equivalentes

recebendo um argumento extra, denominado dicionário [Wadler & Blott, 1989], que armazena, para cada tipo que é instância de uma classe, funções que definem, para este tipo, os símbolos da classe. O trecho de código na Figura 3.4 apresenta o resultado da tradução do código apresentado na Figura 3.1 para o equivalente utilizando dicionários.

```
data Eq a = MkEq (a → a → Bool) (a → a → Bool)

eq (MkEq e _) = e
ne (MkEq _ n) = n

dEqInt :: Eq Int
dEqInt = MkEq primEqInt (λx y → not(primEqInt x y))

dEqBool :: Eq Bool
dEqBool = MkEq f (λx y→ not(f x y))
          where f True True = True
                f False False = True
                f _ _ = False

dEqList :: Eq a → Eq[a]
dEqList d = MkEq el (λx y→ not(el x y))
            where el [] [] = True
                  el (x:xs) (y:ys) = (eq d x y) && (el xs ys)
                  el _ _ = False
```

Figura 3.4. Exemplo de tradução de classes de tipo e instâncias para dicionários.

Pode-se observar, na tradução mostrada na Figura 3.4, que a declaração de classe foi convertida para uma definição de um novo tipo de dados, que representa o dicionário para a classe `Eq`. Este tipo de dados possui um único construtor de valores deste tipo, `MkEq`, que possui como parâmetros dois valores do tipo funcional `a → a → Bool`, que correspondem às funções membro `(==) e (/=)`.

Cada uma das instâncias, definidas na Figura 3.4, foi traduzida para uma função que pode receber dicionários como argumentos (correspondentes a super-classes) e retorna dicionários. A instância que implementa a igualdade para listas, após a tradução, recebe como parâmetro adicional um dicionário para representar as operações da classe `Eq` de seus elementos.

A tradução da função `member` (Figura 3.5), definida originalmente na Figura 3.2, possui um parâmetro extra correspondente ao dicionário da classe `Eq`, que representa a restrição `Eq a` presente em seu tipo original. Além disto, a utilização do símbolo sobrecarregado `(==)` é substituída pela função de projeção `eq`, conforme pode ser observado:

```
member ::  Eq a → a → [a] → Bool
member _ x [] = False
member d x (y:ys) = (eq d x y) || member d x ys
```

Figura 3.5. Tradução da função `member`, utilizando dicionários.

Outra peculiaridade permitida na linguagem Haskell para a definição de classes de tipo é a possibilidade de adicionar implementações *padrão* (*default*) para as funções membro de uma classe. Na definição da classe `Eq` (Figura 3.1), temos definições padrão para os símbolos `(==)` e `(/=)` como sendo:

```
x == y = not (x /= y)
x /= y = not (x == y)
```

Com isto, o programador passa a ter que definir, em instâncias da classe `Eq`, apenas um dos símbolos `(==)` ou `(/=)`, uma vez que a implementação padrão será utilizada para o símbolo omitido.

3.2.2 Ambigüidade

Um problema da abordagem atualmente utilizada por Haskell para o polimorfismo de sobrecarga dependente de contexto é a possibilidade de ocorrência de *ambiguidades*. Considere o seguinte exemplo clássico [Jones, 2002, Hudak et al., 2007]:

```
show :: Show a ⇒ a → String
read :: Read a ⇒String → a

f :: String → String
f s = show (read s)
```

Neste exemplo, `show` converte um valor de qualquer tipo definido como instância da classe `Show` para uma `String`, enquanto `read` faz o inverso para qualquer tipo que é instância da classe `Read`. Considere o tipo da expressão intermediária `(read s)` e suponha existam instâncias das classes `Read` e `Show` para os tipos `Int` e `Bool`, no contexto onde ocorre a definição de `f`. Não há, no contexto desta definição, nenhuma condição que requeira a instanciação destas funções para uma de suas instâncias, ou seja, a instanciação da variável quantificada `a` para um dos tipos `Int` ou `Bool`. O que faz com que o tipo inferido para `f` seja:

```
f ::  (Read a, Show a) ⇒ String → String
```

Tais expressões de tipo são ditas ambíguas em Haskell e são rejeitadas, por conterem um erro. A linguagem Haskell possui a seguinte forma geral de tipos: $\forall\overline{\alpha}.\ \kappa \Rightarrow \tau$, onde $\overline{\alpha}$ é um conjunto de variáveis de tipo e κ são as restrições destas variáveis em relação ao tipo τ. A especificação da linguagem define que um tipo $\forall\overline{\alpha}.\ \kappa \Rightarrow \tau$ de uma expressão é ambíguo se existe alguma variável de tipo presente nas restrições (κ) que não está presente no tipo (τ). Diz-se que tal ocorrência desta variável, neste tipo, é ambígua. Como o tipo inferido para `show(read s)` é `(Read a, Show a).String`, este é considerado ambíguo e é rejeitado pelo compilador.

Uma maneira para contornar esta situação é utilizar *expressões com anotações de tipo*. Por exemplo:

```
f ::  String → String
f s = show ((read s)::  Int)
```

Com a anotação de tipo presente na subexpressão `(read s)::Int`, a definição do símbolo `f` torna-se não ambígua em um contexto onde existam instâncias para o tipo `Int` das classes `Read` e `Show`.

Ambiguidades em operações da classe `Num`[3] são muito comuns. Para evitar a necessidade de tipar toda subexpressão numérica, Haskell adota uma regra bastante *ad hoc*, explicada a seguir, para permitir a eliminação de algumas ambiguidades, baseada no uso de cláusulas *default*, que têm a seguinte forma:

$$\texttt{default}(t_1, \ldots, t_n)$$

onde $n > 0$ e cada t_i deve ser um tipo da classe `Num`. Nas situações onde é detectada uma ambiguidade, uma variável de tipo `v` é instanciável de forma a eliminar a ambigüidade, se (retirado de [Jones, 2002]):

- `v` aparece somente em restrições da forma `C v`, onde `C` é uma classe, e
- pelo menos uma destas classes é uma classe numérica, ou seja, esta é uma subclasse de `Num` ou a própria classe `Num`.

Ocorrências ambíguas de variáveis de tipo são instanciadas para tipos de maneira a eliminar todas as ocorrências ambíguas, se isto for possível. Se houver mais de uma possibilidade para instanciação de variáveis com ocorrências ambíguas, são escolhidos tipos de acordo com a ordem em que estes ocorrem na declaração da cláusula *default* presente no módulo onde ocorreu esta ambigüidade.

[3] A classe `Num` que define o tipo de operações sobre tipos numéricos como adição, subtração, etc.

Somente uma declaração *default* é permitida por módulo e sua visibilidade é restrita ao módulo em que foi definida. Caso nenhuma declaração deste tipo seja fornecida em um módulo qualquer, utiliza-se a seguinte declaração padrão:

```
default(Integer, Double)
```

3.3 Classes de Tipos com Múltiplos Parâmetros

3.3.1 Introdução

A generalização do conceito de classes de tipo, permitindo a definição dessas classes com múltiplos parâmetros, foi originalmente proposta em [Wadler & Blott, 1989]. Considere o seguinte exemplo (transcrito de [Jones, 2000]):

```
class Coerce a b where
   coerce :: a → b
instance Coerce Int Float where
   coerce = convertIntToFloat
```

Diversos artigos incluem exemplos envolvendo o uso de classes de tipo com múltiplos parâmetros (por exemplo, [Jones et al., 1997, Duggan & Ophel, 2002]). Embora várias implementações de Haskell incluam suporte a classes com múltiplos parâmetros, essa extensão ainda não foi incluída na definição oficial da linguagem, por causa de problemas relativos a ambiguidade. Considere por exemplo o seguinte:

```
class Collects a b where
   empty :: b
   insert :: a → b → b
   member :: a → b → Bool
```

Este exemplo foi usado em [Jones, 2000] para descrever uma classe de tipos que define os tipos de operações sobre coleções, onde a variável `a` representa o tipo dos elementos e `b` o construtor da coleção. Pode-se definir como instâncias da classe `Collects`:

- Listas, árvores e outras estruturas de dados que possuem a forma de um construtor aplicado a um tipo.
- Estruturas que utilizam funções de *hashing*.

Possíveis instâncias para esta classe seriam:

```
instance Eq a ⇒ Collects a [a] where ...
instance Ord a ⇒ Collects a (Tree a) where ...
instance (Hashable a, Collects a b) ⇒ Collects a (Array Int b) where
...
```

Neste exemplo ocorre um problema com o tipo da função `empty`. De acordo com a regra de ambigüidade adotada em Haskell (apresentada na Seção 3.2.2), essa função é considerada ambígua, uma vez que seu tipo é `empty::(Collects a b) ⇒ b`.

Uma alternativa para a resolução deste problema é declarar a classe `Collects` como:

```
class Collects a c where
   empty :: c a
   insert :: a →c a → c a
   member :: a →c a → Bool
```

Apesar desta declaração não apresentar problemas de ambiguidade em relação ao símbolo `empty`, ela apresenta o inconveniente de poder ser instanciada apenas para coleções formadas por um construtor de tipos `c` aplicado a um tipo `a`. Para resolver estes problemas de ambiguidades que ocorrem devido ao uso de classes de múltiplos parâmetros, diversas propostas foram elaboradas. Nas próximas seções serão apresentadas estas propostas.

3.3.2 Verificação e Inferência de Tipos

A definição da linguagem Haskell permite apenas que classes com um único parâmetro [Duggan & Ophel, 2002, Jones, 2002] sejam definidas, o que impede o uso de classes de tipos em diversas aplicações práticas. Mesmo diante desta restrição, [Volpano, 1994] mostra que o problema de satisfazibilidade de restrições para tipos polimórficos é $\mathcal{NP}$-difícil e, sem esta restrição, torna-se indecidível.

Nesta seção são apresentadas duas propostas para verificação de tipos para programas que utilizem classes de tipos com múltiplos parâmetros em Haskell sem a necessidade de extensões como dependências funcionais (seção 3.4) e famílias de tipos (seção 3.5). A abordagem proposta neste trabalho é descrita no capítulo 4.

3.3.2.1 A Proposta de Duggan e Ophel

[Duggan & Ophel, 2002] apresenta uma proposta para verificação de tipos na qual são feitas restrições que assemelham-se à utilização de dependências funcionais

[Jones, 2000]. Para isto os autores impõe que toda classe de tipos seja da forma:

$$\texttt{class } \kappa \Rightarrow \texttt{C } \alpha_1, ..., \alpha_m, \beta_1, ..., \beta_n \texttt{ where...}$$

onde as instanciações dos parâmetros α_1, ..., α_m devem unicamente determinar as instanciações dos parâmetros β_1, ..., β_n, para algum m e n declarados pelo programador. Mas, com esta restrição não é possível definir, por exemplo, instâncias como as seguintes:

```
instance (Mult α1 β γ, Add γ γ γ) ⇒
            Mult (Matrix α1) (Matrix β) (Matrix γ) where...
instance (Mult α2 β γ) ⇒ Mult α2 (Matrix β) (Matrix γ) where...
```

uma vez que há a possibilidade de instanciar α_2 como `Matrix` α_1. Para contornar este problema, [Duggan & Ophel, 2002] utilizam restrições que impedem a instanciação de variáveis de tipos. A restrição $\alpha_2 \neq$ `Matrix` δ denota que a variável α_2 não pode ser instanciada para um tipo que possua `Matrix` como seu construtor mais externo. Usando este novo tipo de restrição, temos que o trecho de código anterior é representado como:

```
instance (Mult α2 β γ, α2 ≠ Matrix δ) ⇒
                Mult α2 (Matrix β) (Matrix γ) where...
```

Além disso, os autores propõem utilizar uma estratégia para resolução de sobrecarga que é denominada *resolução de sobrecarga baseada em unificação*[4], definida da seguinte maneira: Dado um conjunto de restrições κ, seja $\kappa\downarrow = (S^r, \kappa^r)$ um par onde S^r é uma substituição e κ^r é um conjunto de restrições que não foram resolvidas após a aplicação de S^r a κ, isto é: $\kappa^r = \{\delta \mid \delta \in S\kappa \wedge tv(\delta) \neq \emptyset\}$. A partir da configuração inicial (id, κ), obtemos $\kappa\downarrow$ pela execução dos seguintes passos:

1. Seja (S, κ) o estado atual do algoritmo, $C\,\overline{\mu} \in \kappa$ uma restrição, $\kappa_1 \Rightarrow C\,\overline{\mu_1}$ uma instância da classe C, $S' = unify(\overline{\mu}, \overline{\mu_1})$ uma substituição que satisfaça todas as restrições $\alpha_i \neq \mu_i \in \kappa$ e $\kappa' = \{\delta \mid \delta \in S\kappa \wedge tv(\delta) \neq \emptyset\} \cup \kappa_1$. Então, a configuração atual do algoritmo passa a ser $(S' \circ S,\ S'\kappa')$.

2. Seja (S, κ) o estado atual do algoritmo, $\kappa_1 \Rightarrow C\,\overline{\mu_1}$ uma instância da classe C e $C\,\overline{\mu_1}$, $C\,\overline{\mu_2} \in \kappa$ restrições tais que cada $\tau_{1i} \in \overline{\mu_1}$ e $\tau_{2i} \in \overline{\mu_2}$ e $\tau_{1\,i} = \tau_{2\,i}$, para todo

[4] tradução livre: *Domain-driven unifying overloading resolution.*

$1 \leq i \leq n$ onde $n \leq k = |\overline{\mu_1}| = |\overline{\mu_2}|$. Suponha que $S' = unifyset(\{(\mu_{1j}, \mu_{2j}) \mid j = n+1, \ldots, k\})$, onde *unifyset* é definido como:

$$unifyset(T) = \begin{cases} id & \text{se } T = \emptyset. \\ S' \circ S & \text{se } T = (\mu_1, \mu_2) \oplus T' \end{cases} \quad \text{onde:} \quad \begin{array}{l} S' = unifyset(S\,T') \\ S = unify(\mu_1, \mu_2) \end{array}$$

unify é uma função que calcula o unificador mais geral de μ_1, μ_2 e $\oplus$ denota a união disjunta de dois conjuntos. Nestas condições, o algoritmo passa a ter como estado atual $(S' \circ S,\ S'\kappa')$ onde $\kappa' = \kappa - \{C\,\overline{\mu_1}\}$.

3. O algoritmo falha caso não exista uma instância que unifique com alguma restrição $C\,\overline{\mu} \in \kappa$.

Para uma melhor compreensão deste algoritmo, consideraremos dois exemplos que exibirão como cada uma das regras funciona. Para a primeira regra, considere o seguinte exemplo:

```
data Employee = E String String Int

class Name a b where name :: a -> b

instance Name Employee String where
    name (E n _ _) = n

f = name (E "J" "a" 2) == name (E "a" "c" 1)
```

No trecho de código anterior, as restrições para `f` são:

```
Eq a, Name Employee a
```

Supondo que todas as definições do prelúdio de Haskell[5] estejam visíveis no ponto da definição de `f`, temos que existem várias instâncias que satisfazem à restrição `Eq a`, porém somente a instância `Name Employee String` satisfaz à restrição `Name Employee a`. Ao unificarmos esta restrição com a instância `Name Employee String` obtemos a seguinte substituição:

$$S = \{\texttt{a} \mapsto \texttt{String}\}$$

[5]Prelúdio (Prelude) é o nome de um módulo que é importado automaticamente por todo módulo em programas Haskell.

que especializa o tipo de `f` para `Bool`, uma vez que no contexto da definição de `f` existem instâncias para `Eq String` e `Name Employee String`.

Para a segunda regra, considere o seguinte exemplo:

```
class Add a b c where (+) :: a -> b -> c


f x y = (x + y, x + y)
```

Neste trecho de código, temos que as seguintes restrições geradas para o tipo de `f` são:

$$\kappa = \{\texttt{Add a b c, Add a b d}\}$$

Pela regra 2, temos que $C\,\overline{\mu_1} =$ `Add a b c`, $C\,\overline{\mu_2} =$ `Add a b d`, onde $\overline{\mu_1} = \{\texttt{a,b,c}\}$, $\overline{\mu_2} = \{\texttt{a,b,d}\}$, $n = 2$ e $k = 3$. Portanto, pela mesma regra, temos que a substituição S' será:

$$S' = \{\mathrm{d} \mapsto \mathrm{c}\} = unifyset(\{(\texttt{c,d})\})$$

o que faz com que a configuração do algoritmo passe a ser igual a

$$(S', S'\,\{\texttt{Add a b d}\}) = (S',\, \{\texttt{Add a b c}\})$$

com isso, temos que o tipo inferido para `f` é:

$$\texttt{f :: Add a b c} \Rightarrow \texttt{a -> b -> c}$$

O algoritmo de inferência e o sistema de tipos proposto por [Duggan & Ophel, 2002] utiliza a resolução de sobrecarga baseada em unificação para prover suporte a classes de tipos com múltiplos parâmetros em Haskell. Um problema desta abordagem é que ela não impõe restrições que garantam a terminação do algoritmo de inferência de tipos. Os autores provam que para programas bem tipados, o algoritmo de inferência sempre termina, mas este pode não terminar para programas com erros de tipos. O programa da figura 3.6 é um exemplo que causa a não terminação do algoritmo de inferência.

Neste trecho de programa, temos que o tipo de `g` possui as seguintes restrições:

$$\kappa = \{\texttt{Foo [a] b, Foo [b] a}\}$$

A regra 1) especifica que a restrição `Foo [a] b` pode ser unificada com a instância `Foo a b` $\Rightarrow$ `Foo [a] [b]` produzindo a seguinte substituição:

$$S' = \{\texttt{b} \mapsto \texttt{[b]}\}$$

```
class Foo a b where
    foo :: a -> b -> Int

instance Foo Int Float where foo x y = 0
instance Foo a b ⇒ Foo [a] [b] where foo (x:_) (y:_) = foo x y

g x y = (foo [x] y) + (foo [y] x)
```

Figura 3.6. Trecho de Código que faz o algoritmo de inferência de [Duggan & Ophel, 2002] não terminar

A nova configuração do algoritmo será formada pela substituição S' e pela aplicação desta ao novo conjunto de restrições κ', que será igual a:

$$\kappa' = \{\delta \mid \delta \in S\,\kappa \wedge tv(\delta) \neq \emptyset\} \cup \kappa_1$$

onde $\kappa_1 = \{\texttt{Foo a b}\}$ e $S = id$. Desta maneira, temos que κ' será igual a:

$$\kappa' = S'\{\texttt{Foo [a] b, Foo a [b]}\} \cup \{\texttt{Foo a b}\}$$

Ao aplicarmos a substituição S' a κ' obteremos o seguinte conjunto de restrições:

$$\kappa' = \{\texttt{Foo [a] [b], Foo a [[b]]}, \texttt{Foo a [b]}\}$$

Com isso temos que o estado do algoritmo será descrito pelo par $(S' \circ id,\, \kappa')$ que será submetido a uma nova iteração do processo de resolução. Novamente, pela regra 1) do algoritmo, temos que a restrição `Foo [a] [b]` unifica com a instância `Foo a b` ⇒ `Foo [a] [b]` produzindo a substituição $S'' = id$ e o seguinte conjunto de restrições:

$$\kappa'' = \{\texttt{Foo a b, Foo [a] [b], Foo a [[b]], Foo a [b]}\}$$

Observe que neste último passo a restrição `Foo a b` é re-inserida no conjunto de restrições, o que faz com que todo processo descrito se repita e cause a não terminação do algoritmo.

Um problema desta abordagem é que este algoritmo é sempre executado durante o processo de inferência para produções *let*. Os autores não especificam nenhum critério para determinar quando o processo de resolução deve ser acionado para um determinado conjunto de restrições.

3.3.2.2 A Proposta de Sulzmann

Outra proposta para verificação / inferência de classes de tipos com múltiplos parâmetros é apresentada em [Sulzmann et al., 2006b]. Neste trabalho os autores estabelecem as seguintes restrições para a decidibilidade do processo de inferência:

- Contextos presentes em definições de instâncias podem somente referenciar varáveis de tipos e em cada restrição todas as variáveis nela contidas devem ser distintas, isto é toda restrição $C\,\overline{\mu}$ deve ser tal que $\neg\exists\,\tau_1, \tau_2 \in \overline{\mu}.\,\tau_1 = \tau_2$.
- Em uma declaração de instância `instance` $\kappa \Rightarrow$ `C` $\overline{\mu}$, pelo menos um tipo τ_i, $1 \leq i \leq |\overline{\mu}|$, não deve ser uma variável e $tv(\kappa) \subseteq tv(\overline{\mu})$.
- Instâncias não devem ser sobrepostas. Para quaisquer duas instâncias:

  `instance` $\kappa_1 \Rightarrow$ `C` $\overline{\mu_1}$

  `instance` $\kappa_2 \Rightarrow$ `C` $\overline{\mu_2}$

  não existe uma substituição S tal que $S\,\overline{\mu_1} = S\,\overline{\mu_2}$.

Se o programa satisfizer estas condições, é provado pelos autores que além de decidível, o processo de inferência produz tipos principais [Sulzmann et al., 2006b]. A prova utiliza uma redução de um programa Haskell contendo classes / instâncias / tipos de dados que satisfazem estas restrições à um programa expresso em *regras de manipulação de restrições*[6] [Frühwirth, 1995]. Tal prova não será apresentada por estar fora do escopo deste trabalho.

3.4 Dependências Funcionais

3.4.1 Introdução

A utilização de classes de tipo com múltiplos parâmetros, apesar de ser uma extensão útil para a definição de símbolos sobrecarregados, facilita o aparecimento de tipos ambígüos. Como exemplo (transcrito de [Hudak et al., 2007]), considere a seguinte tentativa de generalização da classe `Num`:

```
class Add a b r where
   (+) :: a → b → r
instance Add Int Int Int where ...
instance Add Int Float Float where ...
```

[6] do inglês: *Constraint Handling Rules*.

Com isto, permite-se que o programador defina instâncias para somar números de diferentes tipos, escolhendo o tipo do resultado com base no tipo dos argumentos e no contexto no qual a soma ocorre. Apesar desta parecer uma boa solução, ela faz com que expressões simples tenham tipos ambígüos. Como exemplo, considere a seguinte expressão: `n = (x + y) + z`, onde `x`, `y` e `z` são do tipo `Int`. O compilador `GHC` infere o seguinte tipo para `n` (se as opções que desabilitam a restrição de monomorfismo [7] e permitem classes com múltiplos parâmetros forem utilizadas):

```
n :: (Add Int Int a, Add a Int b) ⇒ b
```

Porém, ao adicionarmos a seguinte definição:

```
m = show n
```

resultará em um erro de tipo alertando que não existem instâncias que satisfaçam as restrições:

```
Add Int Int a, Add a Int b
```

Isto ocorre porquê não há uma maneira de especializar a variável de tipo `a` que está presente nas restrições e não no tipo de `n`. Já que tanto `x`, `y` e `z` são do tipo `Int` e existe a instância `Add Int Int Int` pode-se conjecturar que o tipo de `n` será `Int`. Ao anotarmos `n` com o tipo `Int` obtemos o mesmo erro de tipo afirmando que as restrições `Add Int Int a, Add a Int b` não podem ser satisfeitas.

É possível contornar (embora não solucionar) este problema com a utilização de *dependências funcionais* [Jones, 2000]. A idéia é declarar explicitamente uma relação de dependência entre os parâmetros da classe. Utilizando dependências funcionais, a classe `Add` ficaria:

```
Add a b r | a b → r where ...
```

A expressão `a b → r` significa que os tipos `a` e `b` identificam unicamente o tipo `r`. Dependências funcionais são utilizadas para restringir as possíveis instâncias que

[7]

```
f    = show
g    = \x → show x
h::  (Show a) ⇒ a → String
h    = show
main = putStrLn (⊕ '1',⊕ True)
```

A restrição de monomorfismo especifica que a função f não pode ser utilizada no lugar de $\oplus$, enquanto g e h podem. A restrição de monomorfismo proíbe f de possuir um tipo polimórfico restringido. Tal restrição é imposta para evitar tipos ambíguos e a avaliação de expressões onde é esperado que o seu resultado seja compartilhado [Jones, 2002].

um programador pode declarar para uma certa classe de tipos, do mesmo modo que classes de tipos são usadas para restringir os possíveis valores que variáveis de tipo podem assumir em tipos polimórficos. Com isto, a dependência funcional `a b` $\rightarrow$ `r` na classe `Add`, impede a definição das seguintes instâncias:

```
instance Add Int Int Int where   ...
instance Add Int Int Float where ...
```

Isto ocorre porquê os tipos correspondentes às variáveis `a` e `b` não determinam unicamente o tipo de `r`. Nestas duas instâncias as variáveis `a` e `b` assumem o tipo `Int` que não determina unicamente o tipo da variável `r`, que pode assumir os tipos `Int` e `Float`. Para contornar este problema, uma destas instâncias tem que ser removida.

3.4.2 Problemas com Dependências Funcionais

Conforme apresentado anteriormente, dependências funcionais permitem ao programador exercer algum controle sobre o processo de inferência de tipos, pois esta extensão permite a especialização de tipos inferidos com base nas dependências funcionais declaradas pelo programador. Porém, o uso inadequado deste recurso pode levar a comportamentos inesperados, como por exemplo a não terminação do processo de inferência [Sulzmann et al., 2006a]. Tal fato decorre da utilização de dependências funcionais pelo algoritmo de satisfazibilidade de restrições. Como um exemplo deste inconveniente, considere o trecho de código apresentado na figura 3.7. Como `f` utiliza o símbolo

```
class Mult a b c | a b -> c
   (*) ::  a -> b -> c

type Vector b = [b]

instance Mult a b c ⇒ Mult a (Vector b) (Vector c) where
    ...

f b x y = if b then (*) x [y] else y
```

Figura 3.7. Código que causa não terminação da inferência de tipos

sobrecarregado `(*)` em sua definição, a restrição:

```
Mult a (Vector b) b
```

deverá ser adicionada ao tipo de `f`. Porém, ao utilizarmos a dependência `a b -> c` e a instância:

```
instance Mult a b c ⇒ Mult a (Vector b) (Vector c) ...
```

temos que `b = Vector c` para algum `c`. Aplicando a substituição:

$$S = \{\ \texttt{b} \mapsto \texttt{Vector c}\ \}$$

sobre a restrição `Mult a (Vector b) b` obtemos:

```
Mult a (Vector (Vector c)) (Vector c)
```

que pode ser simplificada para `Mult a (Vector c) c` usando a instância declarada no trecho de código da figura 3.7. Mas, a restrição `Mult a (Vector c) c` é idêntica a `Mult a (Vector b) b` a menos do renomeamento de variáveis, o que faz o algoritmo de inferência não terminar.

Devido a possibilidade de não terminação do processo de inferência, restrições devem ser impostas às declarações de classes / instâncias para permitir que o algoritmo de inferência sempre termine. Na seção 3.4.3, serão apresentadas restrições para garantir a corretude e decidibilidade do algoritmo de inferência.

3.4.3 Verificação e Inferência de Tipos

Conforme apresentado anteriormente, dependências funcionais são utilizadas para: restringir o conjunto de possíveis instâncias de uma classe de tipos com múltiplos parâmetros e especialização de tipos durante o processo de inferência. Nesta seção apresentaremos, de maneira suscinta, como dependências funcionais são utilizadas por compiladores Haskell no processo de verificação e inferência de tipos.

3.4.3.1 Restrições para Garantir Corretude e Decidibilidade

Primeiramente, são impostas algumas restrições sobre o formato de classes e instâncias:

- Seja κ um contexto presente em uma classe ou instância. Para cada restrição $C\,\overline{\mu} \in \kappa$ temos que $\overline{\mu}$ deve ser formado apenas por variáveis de tipo e todas estas devem ser distintas.
- Em uma declaração de instância `instance` κ $\Rightarrow$ `C` $\overline{\mu}$ `where...`, pelo menos um dos tipos $\tau_i \in \overline{\mu}$, $1 \leq i \leq |\overline{\mu}|$, não deve ser uma variável de tipo.
- Instâncias não devem ser sobrepostas. Isto é, para quaisquer duas instâncias:

  `instance` κ_1 $\Rightarrow$ `C` $\overline{\mu_1}$ `where...`

  `instance` κ_2 $\Rightarrow$ `C` $\overline{\mu_2}$ `where...`

não existe uma substituição S tal que $S\,\overline{\mu_1} = S\,\overline{\mu_2}$.

Além destas restrições sobre classes e instâncias, [Jones, 2000] apresenta restrições sobre a definição de dependências funcionais:

- **Consistência**: Considere a seguinte declaração de classe `C` e de duas instâncias para esta classe:

  $$\texttt{class } \kappa \Rightarrow \texttt{ C } \overline{\alpha} \texttt{ | } fd_1, \ldots, fd_n$$

  $$\texttt{instance } \kappa_1 \Rightarrow \texttt{ C } \overline{\mu_1} \texttt{ where...}$$
  $$\texttt{instance } \kappa_2 \Rightarrow \texttt{ C } \overline{\mu_2} \texttt{ where...}$$

  Então para cada dependência funcional fd_i, $1 \leq i \leq n$, da forma $\alpha_{i1}, ..., \alpha_{ik} \rightarrow \alpha_{i0}$, temos que a seguinte condição deve ser verdadeira para toda substituição S:

  $$S\,\{\tau_{i1}, ..., \tau_{ik}\} = S\{\tau'_{i1}, ..., \tau'_{ik}\} \rightarrow S\,\tau_{i0} = S\,\tau'_{i0}$$

  onde $\overline{\mu_1} = \{\tau_{i0}, \tau_{i1}, ..., \tau_{ik}\}$ e $\overline{\mu_2} = \{\tau'_{i0}, \tau'_{i1}, ..., \tau'_{ik}\}$.

- **Cobertura**[8]: Considere a seguinte declaração da classe `C`, e uma instância qualquer desta classe:

  $$\texttt{class } \kappa \Rightarrow \texttt{ C } \overline{\alpha} \texttt{ | } fd_1, \ldots, fd_n$$

  $$\texttt{instance } \kappa_1 \Rightarrow \texttt{ C } \overline{\mu_1} \texttt{ where...}$$

  Então para cada dependência funcional fd_i, $1 \leq i \leq n$, da forma $\alpha_{i1}, ..., \alpha_{ik} \rightarrow \alpha_{i0}$, deve ser verdade que:

  $$tv(\tau_{i0}) \subseteq tv(\tau_{i1}, ..., \tau_{ik})$$

  onde onde $\overline{\mu_1} = \{\tau_{i0}, \tau_{i1}, ..., \tau_{ik}\}$.

A restrição de consistência é utilizada para evitar declarações de instâncias inconsistentes, como o exemplo de código da figura 3.8: Neste exemplo temos que a substituição $S = \{\texttt{a} \mapsto \texttt{Float}\}$ viola a condição de consistência, uma vez que ao aplicarmos esta substituição a cada uma das instâncias da figura 3.8 obtemos: `Mult Int Float Float` e `Mult Int Float Int`.

[8] do inglês: *Coverage*.

```
class Mult a b c | a b -> c where
   (*) ::  a -> b -> c

instance Mult Int Float Float where ...  -- (1)

instance Num a ⇒ Mult Int a Int where ...  -- (2)
```

Figura 3.8. Exemplo de instâncias que violam a restrição de consistência

A restrição de cobertura garante que o domínio de uma dependência determina complementamente a imagem desta. Se $\alpha_{i1}, ..., \alpha_{ik} \rightarrow \alpha_{i0}$ é uma dependência funcional, o conjunto $\{\alpha_{i1}, ..., \alpha_{ik}\}$ é o domínio desta dependência e $\{\alpha_{i0}\}$ é a sua imagem. Considere como exemplo, o seguinte trecho de código da figura 3.9.

```
class Mult a b c | a b -> c
   (*) ::  a -> b -> c

instance Mult a b c ⇒ Mult a (Vector b) (Vector c) where
    ...
```

Figura 3.9. Exemplo de instância que viola a condição de cobertura

Suponha que instanciemos os dois primeiros parâmetros da classe `Mult` para `Int` e `Vector Int`. Tal instanciação não determina obrigatoriamente um valor para a variável `c`, já que $\{\texttt{c}\} \not\subseteq tv(\{\texttt{a, Vec b}\})$. A violação da condição de cobertura por uma declaração de instância acarreta a não terminação do algoritmo de verificação / inferência de tipos [Sulzmann et al., 2006a].

As restrições acima não são suficientes para garantir que o processo de inferência de tipos seja correto. Além destas já citadas anteriormente, faz-se necessária a seguinte restrição:

- **Condição sobre Variáveis Ligadas**: Para cada declaração de classe

$$\texttt{class } \kappa \Rightarrow \texttt{C } \overline{\alpha} \texttt{ | } fd_1, \ldots, fd_n$$

deve ser verdade que $tv(\kappa) \subseteq \overline{\alpha}$ e para cada instância

$$\texttt{instance } \kappa_1 \Rightarrow \texttt{C } \overline{\mu_1} \texttt{ where } \ldots$$

deve ser verdade que $tv(\kappa_1) \subseteq tv(\overline{\mu_1})$.

As três restrições (consistência, cobertura e variáveis ligadas) garantem a corretude e terminação do processo de inferência de tipos. A prova desta afirmação está fora do escopo deste trabalho, mas pode ser encontrada em [Sulzmann et al., 2006a].

3.4.3.2 Detecção de Ambüiguidade

Uma expressão e é dita ser semanticamente ambígua se duas ou mais denotações distintas podem ser obtidas para ela, usando sua derivação de tipos [Mitchell, 1996]. De acordo com a definição da linguagem Haskell, um tipo $\forall\overline{\alpha}.\kappa \Rightarrow \tau$ é considerado ambíguo se $\exists v.v \in (tv(\kappa) \cap \overline{\alpha}) \wedge v \notin tv(\tau)$ [Jones, 2002].

Para apresentar como é feita a detecção de ambigüidade utilizando dependências funcionais, primeiramente devemos fazer algumas definições e em seguida as explicaremos por meio de um exemplo. Se F é um conjunto de dependências funcionais, $J \subseteq I$ é um conjunto de índices, então o fechamento de J com respeito a F, J_F^+, é o menor conjunto tal que (onde $X = \{\alpha_1, ..., \alpha_n\}$ e $Y = \{\alpha_0\}$):

- $J \subseteq J_F^+$
- $\forall (X \rightarrow Y) \in F \,.\, X \subseteq J_F^+ \rightarrow Y \subseteq J_F^+$

Intutivamente, J_F^+ é o conjunto de índices que são unicamente determinados pelos índices $i \in J$ ou pelas depedências $(X \rightarrow Y) \in F$. A partir desta definição podemos definir como é feita a detecção de ambigüidade utilizando dependências funcionais. O tipo $\forall\overline{\alpha}.\kappa \Rightarrow \tau$ é considerado ambíguo se $\exists v.v \in (tv(\kappa) \cap \overline{\alpha}) \wedge v \notin (tv(\tau))^+_{F_\kappa}$, onde F_κ é definido como: $F_\kappa = \{tv(\overline{\mu}_X) \rightarrow tv(\overline{\mu}_Y) \mid (C\ \overline{\mu}) \in \kappa \wedge (X \rightarrow Y) \in F_C\}$, $F_C = \{fd_1, ..., fd_n\}$ é o conjunto de dependências funcionais da classe `C` e $\overline{\mu}_X$ representa a projeção da sequência $\overline{\mu}$ sobre X.

Para exemplificar as definições anteriores considere o seguinte trecho de programa:

```
class C a b | a -> b where
    c ::  a -> b

h ::  (C a b) ⇒ a -> Bool
h x = (c x) == (c x)
```

Figura 3.10. Trecho de código contendo um tipo não ambíguo.

De acordo com a definição da linguagem Haskell, o tipo da função `h` é ambíguo uma vez que:

$$\texttt{b} \in tv(\{\texttt{C a b, Eq b}\}) \wedge \texttt{b} \notin tv(\texttt{a -> a})$$

Porém, de acordo com as definições desta seção, o tipo de `h` não é ambíguo [Jones, 2000]. Para entender o porquê, considere o conjunto de restrições sobre o tipo de `h`:

$$\kappa = \{\texttt{C a b, Eq b}\}$$

Informalmente, o tipo

$$\forall\ \texttt{a b.(C a b, Eq b)} \Rightarrow \texttt{a -> Bool}$$

será ambíguo se existir uma variável que seja universalmente quantificada e pertença às restrições presentes em κ e não esteja contida no fechamento das variáveis do tipo simples τ = `a -> Bool` com respeito ao conjunto de dependências $F_{\{\texttt{C a b, Eq b}\}}$. A partir das definições anteriores, temos que o conjunto $F_{\{\texttt{C a b, Eq b}\}}$ será igual a:

$$F_{\{\texttt{C a b, Eq b}\}} = \{\texttt{a -> b}\}$$

e o fechamento deste sobre as variáveis de tipo de τ = `a -> Bool` é:

$$\{\texttt{a}\}^{+}_{F_{\{\texttt{C a b, Eq b}\}}} = \{\texttt{a,b}\}$$

Então de acordo com [Jones, 2000], o tipo

$$\forall\ \texttt{a b.(C a b, Eq b)} \Rightarrow \texttt{a -> Bool}$$

será ambíguo se:

$$\exists v.v \in\ (tv(\kappa) \cap \overline{\alpha}) \wedge v \notin\ (tv(\tau))^{+}_{F_\kappa}$$

mas:

- $\kappa = \{\texttt{C a b, Eq b}\}$ temos que $tv(\kappa) = \{\texttt{a,b}\}$.
- Como τ = `a -> Bool`, temos que $tv(\tau) = \{\texttt{a}\}$ e que $\{\texttt{a}\}^{+}_{F_{\{\texttt{C a b, Eq b}\}}} = \{\texttt{a,b}\}$.

O que nos faz concluir que $tv(\kappa) = \{\texttt{a}\}^{+}_{F_{\{\texttt{C a b, Eq b}\}}}$ e portanto, de acordo com a definição de [Jones, 2000], o tipo de `h` não é ambíguo.

3.4.3.3 Especialização de Tipos

Uma dependência funcional pode ser utilizada de duas formas para especializar tipos inferidos[Jones, 2000]. Seja C uma classe e $X \rightarrow Y \in F_C$, então:

- Suponha que existam duas restrições $C\,\overline{\mu_1}$ e $C\,\overline{\mu_2}$. Se $\overline{\mu_1}_X = \overline{\mu_2}_X$ então $\overline{\mu_1}_Y$ deve ser igual a $\overline{\mu_2}_Y$.
- Suponha que seja inferido uma restrição $C\,\overline{\mu_1}$ e que exista uma instância:

$$\texttt{instance}\ \kappa \Rightarrow C\,\overline{\mu_2}\ \texttt{where}\ \ldots$$

Se $\overline{\mu_1}_X = S\,\overline{\mu_2}_X$, para alguma substituição S, então $\overline{\mu_1}_Y$ e $S\,\overline{\mu_2}_Y$ devem ser iguais.

Em ambos os casos, pode-se usar unificação para garantir que as igualdades citadas sejam satisfeitas. Se a unificação falha, então pode-se concluir que um erro de tipo foi encontrado. Caso contrário, é obtida uma substituição que pode ser utilizada para especializar os tipos inferidos. Estas duas regras para especialização de tipos podem ser aplicadas repetidamente durante o processo de inferência enquanto houverem oportunidades para especialização dos tipos inferidos [Jones, 2000, Jones & Diatchki, 2009].

3.5 Famílias de Tipos

3.5.1 Introdução

Algumas linguagens experimentais como *ATS* [Chen & Xi, 2005], *Cayenne* [Augustsson, 1998] e *Chamaleon* [Sulzmann et al., 2006c] permitem ao programador definir várias formas de funções de tipos, o que os permite escrever programas completos no nível de tipos. Em Haskell, duas extensões ao sistema de tipos permitem expressar computações no nível de tipos[9]: dependências funcionais e famílias de tipos.

Famílias indexadas de tipos [10], ou simplesmente famílias de tipos, são uma extensão ao sistema de tipos de Haskell que permitem a sobrecarga de tipos de dados, isto é, famílias permitem a sobrecarga de tipos de dados da mesma maneira que classes permitem a sobrecarga de funções [Chakravarty et al., 2005b, Chakravarty et al., 2005a].

O conceito de família de tipo pode ser definido formalmente como uma função parcial no nível de tipos, permitindo assim um programa computar quais sobre quais construtores de dados ele irá operar, ao invés de deixá-los fixos estaticamente. Como um primeiro exemplo, considere a seguinte definição de uma família para representar mapeamentos finitos:

Esta definição é análoga a uma definição de classe de tipo convencional, exceto pela presença de uma *família de tipos associada*[11]: `data GMap k :: * -> *`. Observe que esta declaração de família define um nome para esta família (`GMap`), uma variável de tipos e o respectivo *kind*[12] de `GMap k`. Cabe ressaltar que a família `GMap` utiliza como parâmetro a mesma variável que é utilizada pela classe de tipos na qual esta

[9] do inglês: *Type level*

[10] do inglês: *Indexed Type Families*

[11] do inglês: *Associated Type Family*.

[12] *Kinds* classificam tipos da mesma maneira que tipos classificam valores. O *kind* `*` é dito ser o *kind* de tipos. Então, `* -> *` é o *kind* de funções que mapeiam um tipo em outro.

```
class GMapKey k where
   data GMap k :: * -> *
   empty       :: GMap k v
   lookup      :: k -> GMap k v -> Maybe v
   insert      :: k -> v -> GMap k v -> GMap k v
```

Figura 3.11. Definição de mapeamentos finitos usando famílias de tipos.

família foi definida e, portanto, todas as instâncias da família e da classe devem possuir como primeiro parâmetro o mesmo tipo. Como exemplo de uma instância, considere a seguinte implementação que usa como chave do mapeamento um valor inteiro:

```
instance GMapKey Int where
   data GMap Int v        = GMapInt (Data.IntMap.IntMap v)
   empty                  = GMapInt Data.IntMap.empty
   lookup k   (GMapInt m) = Data.IntMap.lookup k m
   insert k v (GMapInt m) = GMapInt (Data.IntMap.insert k v m)
```

Figura 3.12. Uma instância de Família Associada de Tipos.

Na instância definida na figura 3.12, temos que a família `GMap::* -> * -> *`[13] é instanciada com os parâmetros `Int`, que coincide com o parâmetro da classe; e a variável `v` criando o construtor de dados `GMapInt :: IntMap v -> GMap Int v`, que pode ser utilizado para definir funções por casamento de padrão, como por exemplo, as funções `lookup` e `insert` na mesma figura.

Em [Jones, 2000], um dos exemplos para motivar a utilização de dependências funcionais é a definição de uma classe de tipos para representar operações sobre coleções. Este mesmo exemplo, utilizando famílias de tipos, é apresentado na figura 3.13.

No exemplo anterior, foi definida uma família de tipos que relaciona tipos de coleções (representadas pela variável de tipos `c`) com o tipo dos elementos desta coleção. A instância para a família `Elem c` apresentada na figura 3.13, mostra que se o tipo da coleção é `[e]`, então o tipo dos elementos desta é `Elem [e] = e`. Agora, considere a seguinte função `ins`:

```
ins x c = insert x (insert 'y' c)
```

e o respectivo tipo inferido para ela:

[13] Se `GMap k` possui *kind* `* -> *`, temos necessariamente que a variável `k` possui *kind* `*` e o construtor de tipos `GMap * -> * -> *`.

```
type family Elem c
class Collects c where
   empty  :: c
   insert :: Elem c -> c -> c
   member :: c -> [Elem c]

type instance Elem [e] = e
instance Eq (Elem c) => Collects [c] where ...
```

Figura 3.13. Definindo coleções genéricas usando Famílias de Tipos.

```
ins :: (Collects c, Elem c ~ Char) => Elem c -> c -> c
```

Esta função realiza a inserção de um valor `x` em uma coleção `c`, logo depois da inserção do caractere `'y'` nesta mesma coleção. Porém, isto restringe as possíveis instâncias da família `Elem` para que estas sejam iguais a `Char`, o que é representado no tipo de `inst` como `Elem c ~ Char`. Tal restrição é dominada *restrição de igualdade de tipos* [Sulzmann et al., 2007, Schrijvers et al., 2008]. Restrições de igualdade, cuja forma geral é $t_1 \tilde{} t_2$ — que representa que o tipo t_1 deve ser igual a t_2 — podem aparecer nos mesmos pontos da sintaxe que restrições de classe. Portanto, a utilização de famílias de tipos requer o acréscimo deste novo tipo de restrições à linguagem.

3.5.2 Problemas com Famílias de Tipos

Famílias são uma extensão recente [Schrijvers et al., 2008], que visa oferecer as mesmas funcionaliades de dependências funcionais utilizando uma notação "funcional" [Chakravarty et al., 2005b].

O principal problema desta abordagem é a introdução de restrições de igualdade de tipos, que implicam em uma série de dificuldades para a implementação de um algoritmo de satisfazibilidade de restrições para uma relação de equivalência [Jones & Diatchki, 2008]. Como exemplo dos possíveis problemas causados por restrições de igualdade, suponha que durante o processo de inferência a seguinte restrição seja obtida:

```
F a ~ G (F a)
```

onde `F` e `G` são duas famílias de um único parâmetro. Como o operador `~` representa a igualdade entre tipos, podemos substituir `F a` por `G (F a)` em `G (F a)` resultando em `G (G (F a))` que leva um algoritmo de satisfazibilidade a não terminação.

Outro problema relacionado às restrições de igualdade é que estas ainda não estão totalmente a implementadas na versão atual do compilador GHC (versão 6.12.3). A seguinte declaração de classe:

```
class (F a ~ b) ⇒ C a b where
     type F a
```

é rejeitada pelo compilador que fornece uma mensagem de erro afirmando que não provê suporte a restrições de igualdade em contextos de classes. A utilidade de restrições de igualdade em contextos de classes é permitir a conversão direta de classes que utilizem dependências funcionais para classes que utilizem famílias de tipos [Jones & Diatchki, 2008]. O trecho de código anterior é equivalente à mesma classe utilizando dependências funcionais:

```
class C a b | a -> b where ...
```

Isto ocorre porquê a restrição de igualdade (`F a ~ b`) em conjunto com a família associada (`F a`) força que toda instância da classe `C` possua uma instância desta família que satisfaça à restrição de igualdade citada. Se a família (`F a`) deve ser igual a `b`, temos que o valor da variável de tipo `a` determina o valor de `b`, exatamente como a declaração da dependência funcional (`a -> b`).

Na seção 3.5.3 são apresentadas restrições sobre a utilização de famílias de tipos para garantir a terminação do processo de inferência.

3.5.3 Verificação e Inferência de Tipos

Conforme apresentado na seção anterior, famílias de tipos são uma extensão recente ao sistema de tipos de Haskell como uma alternativa às dependências funcionais para uso de classes com múltiplos parâmetros, desenvolvimento de programas em nível de tipos[14] e uso dos chamados tipos de dados algébricos generalizados[15]. Esta seção apresenta informalmente como famílias e restrições de igualdade de tipos são utilizadas durante o processo de verificação / inferência de tipos para Haskell.

3.5.3.1 Restrições para Definição de Famílias de Tipos

Na seção 3.5.2 motivamos a necessidade de restrições sobre a definição de famílias de tipos para garantir a terminação do algoritmo de inferência. Em [Schrijvers et al., 2008]

[14] do inglês: *Type-level programs.*
[15] do inglês: *Generalized Algebraic Data Types.*

é definido como representar instâncias de famílias de tipos como um sistema de reescrita de termos confluente e terminante [Baader & Nipkow, 1998], desde que todas as instâncias atendam as seguintes restrições:

- A cabeça de instâncias de uma família não devem ser sobrepostas. Na declaração de uma instância damos o nome de cabeça a parte da definição que está a esquerda do símbolo "=" e de corpo da instância o lado direito. Como exemplo considere a seguinte instância de uma família `F`:

  ```
  type instance F [a] k = (a,k)
  ```

  neste exemplo temos que a cabeça desta declaração é `F [a] k` e o corpo `(a,k)`.

- O corpo de uma instância de família de tipos não deve possuir aplicações de famílias aninhadas. Observe que com esta restrição não é possível definir a seguinte instância:

  ```
  type instance F a = G (F a)
  ```

  que leva a não terminação do sistema de reescrita associado, uma vez que esta instância gera a seguinte igualdade `F a ~ G (F a)` (seção 3.5.2).

3.5.3.2 Inferência de Tipos

Considere o problema de realizar inferência de tipos no qual o programa (envolvendo famílias de tipos) em questão não possui qualquer tipo de anotação de tipos.

De acordo com [Schrijvers et al., 2008], o algoritmo para inferência de tipos envolvendo famílias de tipos é simples, sendo necessária apenas uma adequação da unificação, que deve *normalizar* os tipos a serem unificados. Porém, deve-se ter certo cuidado ao realizar a unificação para evitar que a normalização de tipos envolvendo famílias produza resultados inesperados[16]. Como exemplo, considere a seguinte expressão:

```
\x -> (insert 'a' x, length c)
```

onde `insert` (definida na figura 3.13) e `length` possuem os seguintes tipos:

```
insert ::  Collects c ⇒ Elem c -> c -> c

length ::  [a] -> Int
```

[16]tradução livre: *get stuck*.

Para inferir o tipo de `\c -> (insert 'c' x, length c)`, inicialmente atribui-se uma nova variável de tipo α para `c`. A chamada da função `insert` faz o algoritmo unificar o tipo `Char` (tipo do parâmetro `'a'`) com o tipo do primeiro parâmetro desta função (`Elem` α), o que produz a restrição `Elem` α `~ Char`. Se neste ponto tentarmos normalizar o tipo `Elem` α não obteremos resultado algum, uma vez que não sabemos que tipo é representado pela variável α. Mas, posteriormente temos que a chamada `length c` força a variável α a ser unificada com `[`β`]`. Com isso temos que a restrição

`Elem` α `~ Char`

passa a ser igual a `Elem [`β`] ~ Char`. Pela instância (definida na figura 3.13):

```
type instance Elem [e] = e
```

temos que a restrição anterior pode ser simplificada para: α `~ Char` e com isto, o tipo da expressão `\x -> (insert 'a' x, length c)` é inferido como:

```
String -> (String, Int)
```

já que o tipo de `insert` é especializado para `Char -> String -> String` devido a restrição de igualdade `Elem` α `~ Char` e da instância definida para a família `Elem`.

3.5.3.3 Verificação de Tipos

Segundo [Schrijvers et al., 2008] os maiores problemas relativos a utilização de famílias de tipos é a verificação de tipos. Estas dificuldades são decorrentes da interação entre instâncias de famílias de tipos, anotações de tipos (com restrições de igualdade) fornecidas pelo programador e casamento de padrões sobre tipos de dados algébricos generalizados.

Conforme citado anteriormente, instâncias de famílias podem ser utilizadas como regras para a reescrita de restrições de igualdade de tipos. O problema de verificação de tipos na presença de famílias é determinar uma solução para um conjunto de restrições de igualdade de tipos que surgem de:

- Anotações de tipos fornecidas pelo programador.
- Casamento de padrão sobre tipos de dados algébricos generalizados.

a partir de um conjunto de regras de reescrita que são geradas pelas instâncias das famílias de tipo em questão. Como estes problemas não estão relacionados diretamente a utilização de classes de tipos com múltiplos parâmetros, não daremos maiores detalhes pois estes problemas estão fora do escopo deste trabalho.

3.6 O Dilema dos Projetistas de Haskell

Um dos temas mais debatidos no processo de padronização de uma nova especificação de Haskell é a adoção de classes de tipos com múltiplos parâmetros. Porém, acredita-se que para isso é necessário adotar uma extensão que permita a utilização destas classes. Até o presente momento, dependências funcionais e famílias de tipos tem sido utilizadas para definir classes de múltiplos parâmetros, o que motiva a seguinte pergunta:

> *A próxima especificação de Haskell deve adotar, como padrão, dependências funcionais ou famílias de tipos?*

Parece haver um consenso de que não é necessário prover suporte a estas duas extensões. Dependências funcionais possuem a vantagem de já serem utilizadas em diversas implementações. Por sua vez, ainda não se tem idéia da expressividade de famílias de tipos. O que leva a seguinte questão:

> *Famílias de tipos e dependências funcionais possuem expressividade equivalente?*

Em [Chakravarty et al., 2005a] é apresentado um argumento informal de que ambas as extensões possuem a mesma expressividade. Mas, como tal afirmativa ainda não possui uma prova formal e ambas introduzem problemas ao já complexo sistema de tipos de Haskell, o dilema até o presente momento permanece sem solução.

3.7 Problemas da Abordagem usada por Haskell para Sobrecarga

Conforme apresentado nas seções anteriores, o sistema de classes de tipo de Haskell permite a definição de símbolos sobrecarregados. Nesta seção são apresentados alguns dos pontos que podem ser vistos como desvantagens desta abordagem para o polimorfismo de sobrecarga:

- A declaração de classes de tipo envolve uma tarefa que não é relacionada com a resolução da sobrecarga: a de agrupar logicamente nomes em uma mesma construção da linguagem.

- Para qualquer nova definição de qualquer símbolo sobrecarregado o tipo desta nova implementação deve ser uma instância do tipo declarado em sua declaração de classe, ou esta deve ser modificada.

- Mesmo para um símbolo sobrecarregado para o qual todas instâncias são conhecidas *a priori*, deve-se obrigatoriamente declarar uma classe de tipo que possua a definição do tipo mais geral para a operação sobrecarregada em questão.

Maiores detalhes sobre problemas relacionados a abordagem de polimorfismo de sobrecarga de Haskell podem ser encontrados em [Camarao & Figueiredo, 1999b].

3.8 Conclusão

Neste capítulo foram apresentadas as principais características de Haskell para o suporte a sobrecarga. Apresentamos os problemas relacionados a ambiguidade introduzida por classes de tipos com múltiplos parâmetros e as principais extensões propostas para sua utilização. Para cada uma destas extensões, foram apresentados exemplos de sua utilização e foram descritas de maneira informal as restrições e os problemas decorrentes da adoção de cada uma destas.

No próximo capítulo é apresentado o sistema proposto por este trabalho que visa resolver o dilema para a adoção de classes com múltiplos parâmetros. Este sistema não adiciona nenhuma complexidade adicional ao sistema de tipos de Haskell, o que acarreta uma implementação mais simples de um algoritmo para inferência de tipos para esta linguagem.

Capítulo 4

O Sistema de Tipos Proposto

4.1 Introdução

No capítulo 3 foram apresentados as principais características da abordagem de Haskell para sobrecarga e seus principais problemas. Neste capítulo é apresentada a definição formal do sistema de tipos proposto neste trabalho que visa solucionar, de maneira simples, o principal problema relacionado a adoção de tipos com múltiplos parâmetros, a saber, ambiguidade de expressões que usam nomes sobrecarregados introduzidos em classes com múltiplos parâmetros.

Este capítulo é organizado da seguinte maneira. Primeiramente é formalizada a sintaxe da linguagem núcleo considerada. Na seção 4.3, são apresentadas algumas propriedades de substituições, a seção 4.4 discorre sobre contextos de tipos e a seção 4.5 sobre ordens parciais sobre tipos e substituições, e é apresentado e formalizado um algoritmo capaz de calcular a *menor generalização comum* de um conjunto de tipos [Camarao & Figueiredo, 1999a]. A seção 4.7 apresentará detalhes sobre o algoritmo para satisfazibilidade de restrições. Finalmente, a seção 4.8 apresenta o sistema de tipos proposto, seu algoritmo de inferência, detalhes de implementação de um *front-end* para um compilador Haskell que o utiliza e suas propriedades de tipagem principal e tipo principal.

4.2 Sintaxe

4.2.1 Termos

A sintaxe dos termos da linguagem núcleo consiste basicamente de *core-ML* [Milner, 1978, Damas & Milner, 1982], mas chamaremos esta linguagem de *core-*

Haskell para enfatizar a existência de um contexto global que possui todas as definições de classes e instâncias (maiores detalhes na seção 4.4).

Os termos da linguagem são definidos pela seguinte gramática:

$$\begin{array}{lll} \text{Variáveis de termos (símbolos)} & x \in \mathbf{X} & \\ \text{Expressões} & e \in \mathbf{E} & ::= \quad x \mid \lambda x.\, e \mid e\, e' \mid \texttt{let}\ x = e\ \texttt{in}\ e \end{array}$$

Figura 4.1. Sintaxe Livre de Contexto da Linguagem *Core-Haskell*.

4.2.2 Sintaxe de Tipos e Kinds

Tipos e *kinds* são expressos utilizando a seguinte gramática:

$$\begin{array}{lll} \text{Kind} & k \in \mathbf{K} & ::= \star \mid k \to k' \\ \text{Expressão de Tipo Simples} & \mu \in \mathbf{T} & ::= \alpha \mid T \mid \mu_1\, \mu_2 \\ \text{Tipo Simples} & \tau & \equiv \mu \\ \text{Restrição} & \delta \in (\mathbf{C} \times \overline{\mathbf{T}}) & ::= C\, \overline{\mu} \\ \text{Tipos} & \sigma \in \Sigma & ::= \tau \mid \kappa \Rightarrow \tau \mid \forall \alpha.\, \sigma \end{array}$$

$$\begin{array}{llll} \text{Nome de Classe} & C \in \mathbf{C} & \text{Construtor de Tipos} & T \\ \text{Variável de Tipo} & \alpha, \beta \in \mathbf{V} & \text{Conjunto de Restrições} & \kappa \in \mathcal{P}(\mathbf{C} \times \overline{\mathbf{T}}) \end{array}$$

Figura 4.2. Sintaxe livre de contexto de tipos

Cabe ressaltar que o nome *variável de tipo* é usado por simplicidade. Na verdade, trata-se de variáveis de expressões de tipo.

4.2.2.1 Tipos Simples e Kinds

Um *kind* é uma propriedade de expressões de tipos simples. O *kind* de variáveis de tipos e o de construtores de tipos são dados, respectivamente, pelas funções $\text{kind}_{\mathbf{V}} : \mathbf{V} \to \mathbf{K}$ e $\text{kind}_{\mathbf{C}} : \mathbf{C} \to \mathbf{K}$. Estas funções induzem famílias indexadas por *kind* de variáveis de tipo e de construtores, especificadas da seguinte maneira:

$$\begin{array}{rcl} \mathbf{V}^k & = & \{\alpha \in \mathbf{V} \mid \text{kind}_{\mathbf{V}}(\alpha) = k\} \\ \mathbf{C}^k & = & \{C \in \mathbf{C} \mid \text{kind}_{\mathbf{C}}(C) = k\} \end{array}$$

Utilizamos um índice superior k para identificar o *kind* de variáveis de tipo e construtores:

$$\begin{array}{rcl} \alpha^k & \in & \mathbf{V}^k \\ C^k & \in & \mathbf{C}^k \end{array}$$

O *kind* de expressões de tipo simples é dado pela função parcial $kind_{\mathbf{T}} : \mathbf{T} \to K$, definida como:

$$\text{kind}_{\mathbf{T}} = \begin{cases} k & \text{se } \tau = \alpha^k \text{ ou } \tau = C^k \\ k & \text{se } \tau = \tau_1\,\tau_2,\ \text{kind}_{\mathbf{T}}(\tau_1) = k' \to k \text{ e } kind_{\mathbf{T}}(\tau_2) = k' \end{cases}$$

Consideraremos somente expressões de tipo que forem *bem formadas*, isto é, aquelas que tem um *kind* definido. Por isso, o conjunto $\mathbf{T}$ só irá incluir expressões bem formadas. Por definição, o *kind* de tipos simples é $\star$.

Como usual, expressões de tipo e de *kind* que envolvem o construtor $(\to)$ são escritas de forma infixa:

$$\mu_1 \to \mu_2 \equiv ((\to)\,\mu_1)\,\mu_2$$
$$k_1 \to k_2 \equiv ((\to)\,k_1)\,k_2$$

onde μ_1 e μ_2 são duas expressões de tipos e k_1, k_2 duas expressões de *kind*.

A notação $[\mu]$ é usada para expressar a aplicação do construtor de listas (que possui *kind* $\star \to \star$) ao tipo μ:

$$[\mu] \equiv ([\,])\,\mu$$

O conjunto de variáveis de tipo de uma expressão de tipo simples é dado pela função $tv : \mathbf{T} \to \mathcal{P}(\mathbf{V})$, definida como:

$$tv(\mu) = \begin{cases} \alpha & \text{se } \mu = \alpha \\ \emptyset & \text{se } \mu = C \\ tv(\mu_1) \cup tv(\mu_2) & \text{se } \mu = \mu_1\,\mu_2 \end{cases}$$

Esta operação pode ser utilizada para denotar a família de variáveis de um conjunto $\mathbf{J}$ de expressões de tipo:

$$tv(\mathbf{X}) = \bigcup\{tv(\mu) \mid \mu \in \mathbf{J}\}$$

4.2.2.2 Restrições de Classe

Conforme apresentado na seção 3.2, tipos polimórficos em Haskell podem conter restrições que limitam as possíveis instanciações de variáveis quantificadas neste tipo. Uma restrição δ consiste de um par $(C, \overline{\mu})$, onde C é um nome de classe e $\overline{\mu}$ uma sequência de tipos simples, cada tipo simples correspondendo ao parâmetro da classe de mesma ordem na sequência em $(C, \overline{\mu})$ (se $\overline{\mu} = \{\mu_1, ..., \mu_n\}$ então μ_1 corresponde ao primeiro parâmetro, μ_2 ao 2^o parâmetro, ..., μ_n ao n-ésimo parâmetro da classe C). Por questão de simplicidade, representaremos o par $(C, \overline{\mu})$ como $C\ \overline{\mu}$.

O conjunto de variáveis de tipos de uma restrição é dado pela função:

$$tv : (\mathbf{C} \times \overline{\mathbf{T}}) \rightarrow \mathcal{P}(\mathbf{V})$$

onde $tv(C\ \overline{\mu}) = tv(\overline{\mu})$. A mesma função pode ser estendida para conjuntos de restrições de maneira simples:

$$tv(\kappa) = \bigcup_{1 \leq i \leq n} tv(\mu_i) \text{ para } \kappa = \{C_1\ \overline{\mu_1}, ..., C_n\ \overline{\mu_n}\} \text{ e } n \geq 0.$$

4.2.2.3 Tipos

Uma expressão $\sigma = \forall\overline{\alpha}.\kappa.\tau$ denota um tipo. Se $\kappa = \emptyset$, dizemos que este é um tipo irrestrito, caso contrário dizemos que é restringido. Caso $\overline{\alpha} = \emptyset$, dizemos que este é um tipo monomórfico, caso contrário dizemos que é um tipo polimórfico. Para simplificar a sintaxe, usamos as seguintes abreviações:

- $\forall\overline{\alpha}.\tau$, quando $\kappa = \emptyset$;
- $\kappa.\tau$, quando $\overline{\alpha} = \emptyset$;
- τ, quando $\kappa = \overline{\alpha} = \emptyset$.

O conjunto de variáveis livres de um tipo é dado pela função $tv : \Sigma \rightarrow \mathcal{P}(\mathbf{V})$, definida a seguir:

$$tv(\forall\overline{\alpha}.\kappa.\tau) = (tv(\kappa) \cup tv(\tau)) - \overline{\alpha}$$

onde $\overline{\alpha} = \{\alpha_1, \alpha_2, ..., \alpha_n\}$.

Novamente, esta função pode ser estendida para conjuntos de tipos da seguinte maneira:

$$tv(\mathbf{X}) = \bigcup\{tv(\sigma) \,|\, \sigma \in \mathbf{X}\}$$

4.3 Substituições

Uma substituição S é uma função de tipo para expressões de tipo simples. O conjunto de todas as substituições é denotado por $\mathbf{S}$ e a substituição identidade é denotada por *id*.

Define-se o domínio de uma substituição S, $\mathrm{dom}(S)$, como o conjunto de variáveis $\alpha \in \mathbf{V}$ para as quais $S(\alpha)$ é distinto de α:

$$\mathrm{dom}(S) = \{\alpha \,|\, S(\alpha) \neq \alpha\}$$

Neste trabalho consideramos apenas substituições para as quais o conjunto dom(S) é finito e que preservam o *kind* das variáveis, i.e. $S(\alpha^k) \in T^k$ para qualquer *kind* $k \in \mathbf{K}$.

Toda substituição S pode ser estendida para um homomorfismo sobre expressões de tipo simples usando o operador $(\ )^*$, definido como:

$$\begin{array}{lcl} S^*(\alpha) & = & S(\alpha) \\ S^*(C) & = & C \\ S^*(\mu_1\,\mu_2) & = & S^*(\mu_1)\,S^*(\mu_2) \end{array}$$

Para evitar o excesso de notação, escreveremos $S(\mu)$ ao invés de $S^*(\mu)$, e $S\mu$ ao invés de $S(\mu)$. O termo $[\alpha \mapsto \mu]$ denota a substituição definida como:

$$[\alpha \mapsto \mu](\beta) = \begin{cases} \mu & \text{se } \alpha = \beta \\ \beta & \text{se } \alpha \neq \beta \end{cases}$$

Uma substituição pode ser especificada por um conjunto de pares $(\alpha \mapsto \mu)$. Para $n \geq 1$ e para $\alpha_i \neq \alpha_j$, com $1 \leq i < j \leq n$, a notação $[\alpha_1 \mapsto \mu_1, ..., \alpha_n \mapsto \mu_n]$ denota a substituição S tal que:

$$S(\alpha) = \begin{cases} \mu_i & \text{se } \alpha = \alpha_i \\ \alpha & \text{caso contrário} \end{cases}$$

Substituições podem ser estendidas usando homomorfismos para o domínio de restrições $(S\,\delta)$, conjuntos de restrições $(S\,\kappa)$ e tipos restringidos $(S(\kappa.\tau))$, i.e. aplicando a substituição a cada variável de tipo que aparece como subtermo. A extensão de uma substituição para uma função de conjuntos de tipos para conjuntos de tipos é definida como:

$$S(X) = \bigcup\{S\mu \,|\, \mu \in X\}$$

onde X é um conjunto de expressões de tipos.

Uma substituição também pode ser estendida para uma operação sobre tipos polimórficos, mas é necessário cuidado para evitar a captura de variáveis ligadas. A aplicação de uma substituição S a um tipo σ é dita *livre de captura* se $tv(S\sigma) = tv(S(tv(\sigma)))$.

A função tv é estendida para substituições conforme a seguinte definição:

$$tv(S) = dom(S) \cup \{tv(S(\alpha)) \,|\, \alpha \in\ dom(S)\}$$

4.4 Contextos de Tipos

Um contexto de tipos é conjunto que contém informações sobre um programa, que podem ser utilizadas pelas regras do sistema de tipos. Neste trabalho faremos a distinção entre três contextos distintos, a saber:

- $\Gamma^s \subseteq \mathbf{X} \times \mathbf{T}$: Contexto formado pelas suposições de tipos para símbolos sobrecarregados ou não. Veja explicação no item seguinte.

- $\Gamma^{cls} \subseteq \mathbf{C} \times \overline{\mathbf{V}} \times \mathcal{P}(\mathbf{C} \times \overline{\mathbf{T}})$: Contexto formado pelas restrições de classes definidas no programa. Cada classe

$$\begin{aligned} &\texttt{class } \forall\overline{\alpha}.\kappa \Rightarrow C\overline{\alpha} \texttt{ where} \\ &\quad x_1 :: \kappa_1 \Rightarrow \tau_1 \\ &\quad\quad \vdots \\ &\quad x_n :: \kappa_n \Rightarrow \tau_n \end{aligned}$$

  introduz uma *restrição de classe* $(C, \overline{\alpha}, \kappa)$ em Γ^{cls}. Por simplicidade, a tripla $(C, \overline{\alpha}, \kappa)$ será representada como: $\forall\overline{\alpha}.\kappa \Rightarrow C\,\overline{\alpha}$.

  Além desta restrição em Γ^{cls}, esta declaração de classe introduz suposições de tipos em Γ^s: $x_i :: \sigma_i$ onde $\sigma_i = \forall\overline{\alpha_i}.\kappa \cup \kappa_i \Rightarrow \tau_i$ e $\overline{\alpha_i} = tv(\kappa \cup \kappa_1 \Rightarrow \tau_i)$.

  Adicionalmente, definiremos $\Gamma^{cls}(C) = \forall\overline{\alpha}.\kappa \Rightarrow C\,\overline{\alpha}$ e $\Gamma^{cls}(x_i) = \sigma_i$ ($\Gamma^{cls}(x)$ é indefinido se x não é um símbolo sobrecarregado).

- $\Gamma^{ins} \subseteq \mathbf{C} \times \overline{\mathbf{T}} \times \mathcal{P}(\mathbf{C} \times \overline{\mathbf{T}})$: Contexto formado pelas restrições de instâncias definidas no programa. Cada instância:

$$\begin{aligned} &\texttt{instance } \kappa \Rightarrow C\overline{\mu} \texttt{ where} \\ &\quad x_1 = e_1 \\ &\quad\quad \vdots \\ &\quad x_n = e_n \end{aligned}$$

  introduz uma *restrição de instância* $\forall\overline{\alpha}.\kappa \Rightarrow C\,\overline{\mu}$ em Γ^{ins}. Para restrições de instâncias, também utilizaremos uma notação similar a já adotada para tipos. Denotaremos por $\Gamma^{ins}(C)$ o conjunto de restrições de instâncias definidas para uma classe C.

O contexto de tipos do escopo mais externo de um programa (escopo global),Γ_o, é a união de cada um destes contextos, i.e. $\Gamma_o = \Gamma^s \cup \Gamma^{cls} \cup \Gamma^{ins}$.

Nas definições anteriores não foi imposta nenhuma restrição sobre as instâncias. Porém, consideraremos que Γ^{ins} contém apenas restrições de instância bem formadas. Antes de introduzirmos a definição de instâncias bem formadas na seção 4.4.2, apresentaremos a definição de casamento de classes-instâncias.

4.4.1 Casamento de Classes-Instâncias

A operação de casamento entre restrições de classes e instâncias visa encontrar uma substituição S que torne as cabeças de uma classe $(C\,\overline{\alpha})$ e de uma instância $(C\,\overline{\mu})$ iguais (i.e. $S(C\,\overline{\alpha}) = C\,\overline{\mu}$) e um conjunto κ de restrições de super-classes de C.

$$\boxed{\Gamma \models^{cls} \delta\,[S,\kappa]}$$

$$\frac{\Gamma^{cls}(C) = \kappa \Rightarrow C\overline{\alpha} \qquad S(C\overline{\alpha}) = C\overline{\mu}}{\Gamma \models^{cls} C\overline{\mu}\,[S,\kappa]}$$

Figura 4.3. Casamento de Classes-Instâncias

4.4.2 Instâncias Bem Formadas

Informalmente, uma restrição de instância $\forall\overline{\alpha}.\kappa \Rightarrow C\,\overline{\mu}$ é bem formada em Γ^{ins} se as seguintes condições são verdadeiras:

- $tv(\kappa) \subseteq tv(C\,\overline{\mu})$
- A cabeça da instância $(C\,\overline{\mu})$ deve casar com a correspondente restrição de classe, i.e. $\Gamma \models^{cls} \delta\,[S,\kappa]$ para algum S e algum κ.
- O contexto κ deve ser bem formado — i.e., cada restrição $C'\overline{\mu}' \in S\,\kappa$, onde $\Gamma \models^{cls} \delta\,[S,\kappa]$, deve ocorrer em κ, exceto se $C'\overline{\mu}'$ não possui variáveis de tipo, o que torna necessária a existência de uma declaração de uma instância que satisfaça esta restrição.

As condições anteriores são expressas formalmente pela regra presente na figura 4.4.

4.4.2.1 Exemplos

Para tornar mais claras as definições anteriores, vamos considerar o seguinte trecho de código de exemplo:

$$\boxed{\Gamma \models^{wfi} \kappa \Rightarrow C\,\overline{\mu}}$$

$$\frac{\begin{array}{l} tv(\kappa) \subseteq tv(C\,\overline{\mu}) \\ \Gamma \models^{cls} C\,\overline{\mu}\,[S_C, \kappa_C] \\ S_C\,\kappa_C - \kappa_0 \subseteq \kappa \;\text{ onde }\; \kappa_0 = \{\delta \in S_C\,\kappa_C \mid tv(\delta) = \emptyset\} \\ \text{para cada } C'\,\overline{\mu}' \in \kappa_0, C'\,\overline{\mu}' \in \Gamma^{ins}(C') \end{array}}{\Gamma \models^{wfi} \kappa \Rightarrow C\,\overline{\mu}}$$

Figura 4.4. Restrição de Instância Bem Formada

```
class A α  where  ...
class B α β  where  ...
class C α  where  ...
class {A α, B α β} ⇒ D α β γ  where  ...
instance A Int  where  ...
instance {B Int β, C γ} ⇒ D Int β γ  where  ...
```

Figura 4.5. Código de Exemplo para a Definição de Instâncias Bem Formadas

A instância para a classe D é bem formada no contexto Γ que inclui todas as definições presentes na figura 4.5, já que: 1) $tv(\{B\,\texttt{Int}\,\beta,\ C\,\gamma\}) \subseteq tv(D\,\texttt{Int}\,\beta\,\gamma)$; 2) $\Gamma \models^{cls} D\,\mathrm{Int}\,\beta\,\gamma\,[S_D, \kappa_D]$, onde $S_D = [\alpha \mapsto \mathrm{Int}]$ e $\kappa_D = \{A\,\alpha,\ B\,\alpha\,\beta\}$; 3) o contexto para esta declaração de instância inclui a restrição $B\,\mathrm{Int}\,\beta$ (que corresponde a $B\,\alpha\,\beta \in \kappa_D$), mas não inclui a restrição $A\,\texttt{Int}$ (correspondente a $A\,\alpha \in \kappa_D$), que não contém variáveis de tipo e possui uma declaração de instância correspondente. Observe que o contexto de uma declaração de instância pode também incluir restrições que não ocorrem em sua declaração de classe, como a restrição $C\,\gamma$ na instância da classe D.

4.5 Ordens Parciais

Esta seção define ordens parciais e seus semi-reticulados correspondentes [Davey & Priestly, 1990] para um conjunto de expressões de tipos simples, substituições e contextos de tipos. Tais definições serão utilizadas para a a demonstração das propriedades de Tipo e Tipagem Principal (definidas na seção 4.8.5) e na definição de uma função para computar a menor generalização comum de um conjunto de tipos ou substituições [Camarao & Figueiredo, 1999a].

4.5.1 Expressões de Tipos Simples

4.5.1.1 Pré-Ordem de Expressões de Tipos Simples

A relação $\preceq_{\mathbf{T}}$ é uma pré-ordem sobre o conjunto de tipos simples, definida como:

$$\mu \preceq_{\mathbf{T}} \mu' \text{ se e somente se } S\mu = S\mu', \text{ para algum } S.$$

Se $\mu \preceq_{\mathbf{T}} \mu'$ e $\overline{\alpha} = tv(\mu)$ e $\overline{\alpha}' = tv(\mu')$, dizemos que $\forall\overline{\alpha}'.\mu'$ é mais geral do que $\forall\overline{\alpha}.\mu$, ou que é uma generalização de $\forall\overline{\alpha}.\mu$.

4.5.1.2 Equivalência Módulo Renomeamento de Variáveis

A partir da pré-ordem $\preceq_{\mathbf{T}}$ podemos definir a relação de equivalência $\equiv_{\mathbf{T}}$ como:

$$\mu \equiv_{\mathbf{T}} \mu' \text{ se e somente se } \mu \preceq_{\mathbf{T}} \mu' \text{ e } \mu' \preceq_{\mathbf{T}} \mu$$

Se $\mu \equiv_{\mathbf{T}} \mu'$, dizemos que μ é equivalente a μ' módulo renomeamento de variáveis.

4.5.1.3 Ordem Parcial de Expressões de Tipos Simples

Seja $T_{\equiv}$ o conjunto de classes de equivalência de $\equiv_{\mathbf{T}}$, podemos estender a pré-ordem $\preceq_{\mathbf{T}}$ para uma ordem parcial sobre $T_{\equiv}$. Denotaremos por $[\mu]_{\equiv}$ a classe de equivalência de uma expressão de tipo simples μ módulo $\equiv_{\mathbf{T}}$ ($[\mu]_{\equiv} = \{\mu' \mid \mu \equiv_{\mathbf{T}} \mu'\}$) definimos a ordem parcial $\leq_{\mathbf{T}}$ como:

$$[\mu]_{\equiv} \leq_{\mathbf{T}} [\mu']_{\equiv} \text{ se e somente se } \mu' \preceq_{\mathbf{T}} \mu$$

O elemento $[\alpha]$, onde α é uma variável de tipo qualquer, possui a propriedade de ser o *elemento máximo* (*majorante*) da ordem parcial, i.e. para todo μ, $[\mu] \leq_{\mathbf{T}} [\alpha]$.

Como qualquer elemento de uma classe de equivalência pode ser usado para representar a classe, omitiremos os colchetes e o sinal $\equiv$ quando nos referirmos a elementos de $\mathbf{T}_{\equiv}$, escrevendo μ ao invés de $[\mu]_{\equiv}$.

4.5.1.4 Semi-Reticulado de Expressões de Tipos Simples

Dizemos que $[\mu] = [\mu_1] \vee [\mu_2]$, ou que $\mu = \mu_1 \vee \mu_2$ (μ é o supremo de μ_1 e μ_2), se μ for mais geral que μ_1 e μ_2, e se for o elemento menos geral com esta propriedade.

O supremo de um conjunto finito não-vazio $X \subseteq \mathbf{T}_{\equiv}$ ($\bigvee X$), é definido como:

$$\bigvee X = \begin{cases} \mu & \text{se } X = \{\mu\} \\ \mu \vee \mu' & \text{se } X = \{\mu\} \cup X' \text{ e } \mu' = \bigvee X' \end{cases}$$

A tupla ($\mathbf{T}_{\equiv}$,$\bigvee$,[α]) forma um semi-reticulado com as seguintes propriedades (que serão provadas posteriormente):

- Para todo $\mu_1, \mu_2 \in \mathbf{T}_{\equiv}$, existe $\mu = \mu_1 \vee \mu_2$. Isto é, todo par de elementos de $\mathbf{T}_{\equiv}$ possui supremo.
- Para todo $X \subseteq \mathbf{T}_{\equiv}$, existe $\bigvee X$, i.e. todo subconjunto de $\mathbf{T}_{\equiv}$ possui supremo.
- Toda cadeia crescente[1] de $(\mathbf{T}_{\equiv}, \leq_{\mathbf{T}})$ é finita.

Para exemplificar, a figura 4.6 descreve um fragmento do semi-reticulado de expressões de tipo simples. Note que o tipo simples $\alpha \to \alpha$ é o supremo de Int $\to \alpha$ e $\alpha \to$ Int, enquanto que o tipo simples $\alpha_1\ \alpha_2$. é o supremo dos tipos simples Int$\to$ Int e [Int].

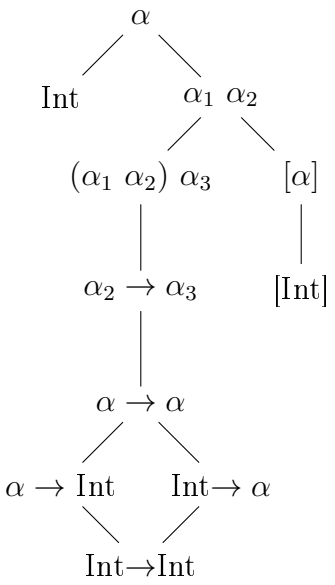

Figura 4.6. Fragmento de Semi-Reticulado de Tipos Simples

4.5.2 Substituições

4.5.2.1 Pré-Ordem de Substituições

A pré-ordem $\preceq_{\mathbf{S}}$ sobre o conjunto de substituições **S** é definida como:

$$S_1 \preceq_{\mathbf{S}} S_2 \text{ se, e somente se existe } S \text{ tal que } S \circ S_2 = S_1$$

[1] Uma cadeia crescente é uma sequência $\{a_1, a_2, ..., a_n\}$ de elementos de um conjunto parcialmente ordenado $(P, \leq)$ tais que $a_1 < a_2 < ... < a_n$.

A pré-ordem $\preceq_{\mathbf{S}}$ induz uma relação de equivalência, $\equiv_{\mathbf{S}}$, definida como:

$$S_1 \equiv_{\mathbf{S}} S_2 \text{ se, e somente se, } S_1 \preceq_{\mathbf{S}} S_2 \text{ e } S_2 \preceq_{\mathbf{S}} S_1$$

4.5.2.2 Ordem Parcial de Substituições

Tomando $S_{\equiv}$ como o conjunto de classes de equivalência de substituições módulo $\equiv_{\mathbf{S}}$, e $[S]_{\equiv}$ como a classe de equivalência da substituição S, podemos definir a ordem parcial $\leq_{\mathbf{S}}$ sobre $S_{\equiv}$ da seguinte forma:

$$[S]_{\equiv} \leq_{\mathbf{S}} [S']_{\equiv} \text{ se, e somente se, } S \preceq_{\mathbf{S}} S'$$

Se $S \preceq_{\mathbf{S}} S'$, dizemos que S' é mais geral (ou uma generalização de S), ou ainda que S é mais específico que S' (ou uma especialização de S').

O elemento máximo (majorante) desta ordem parcial é a substituição identidade (id).

Novamente, serão omitidos os colchetes e o símbolo $\equiv$ quando nos referirmos a elementos de $S_{\equiv}$, escrevendo S ao invés de $[S]_{\equiv}$.

4.5.2.3 Semi-Reticulado de Substituições

Da mesma maneira que definimos o semi-reticulado dos tipos simples, podemos definir o semi-reticulado de substituições ($S_{\equiv}$,$\bigvee$,id), que também possui as mesmas propriedades citadas na seção 4.5.1.4.

A próxima seção apresentará o algoritmo para calcular o supremo (menor generalização comum) de tipos e substituições.

4.5.3 Tipos

Em sistemas de tipos com suporte a polimorfismo paramétrico, a ordenação de tipos é tal que $\forall\alpha.\sigma \leq_{\sigma} [\overline{\alpha} \mapsto \overline{\mu}]\sigma$, para todos os tipos simples μ. Porém, quando consideramos tipos restringidos, devemos levar em conta as restrições presentes nestes tipos [Faxén, 2003, Camarao & Figueiredo, 1999a].

4.5.3.1 Ordem Parcial de Tipos

Seja Σ o conjunto de todos os tipos. Podemos definir a ordem parcial $\leq_{\Sigma}$ sobre Σ da seguinte maneira:

$\sigma_1 \leq_{\Sigma} \sigma_2$ se, e somente se existe $[\overline{\alpha} \mapsto \overline{\mu}]$, tal que $[\overline{\alpha} \mapsto \overline{\mu}]\sigma_2 = \sigma_1$

onde $\sigma_2 = \forall\overline{\alpha'}.\kappa_2 \Rightarrow \tau$ e $\overline{\alpha} \subseteq \overline{\alpha'}$.

4.5.4 Contextos de Tipos

4.5.4.1 Ordem Parcial de Contextos de Tipos

A relação $\leq_\Gamma$ é uma ordem parcial sobre o conjunto de contextos de tipos, definida como:

$$\Gamma_1 \leq_\Gamma \Gamma_2 \text{ se e somente se para toda variável } x, \text{ temos que } \Gamma_1(x) \leq_\Sigma \Gamma_2(x).$$

4.5.5 Tipagens

Sejam σ e Γ serem um tipo e um contexto de tipos respectivamente. Dá-se o nome de *tipagem* ao par (σ, Γ).

4.5.5.1 Ordem Parcial de Tipagens

A relação $\leq_\omega$ é uma ordem parcial sobre o conjunto de tipagens, definida como:

$$(\sigma_1, \Gamma_1) \leq_\omega (\sigma_2, \Gamma_2) \text{ se e somente se } \Gamma_1 \leq_\Gamma \Gamma_2 \text{ e } \sigma_1 \leq_\Sigma \sigma_2.$$

As definições das ordens parciais de tipos, contextos de tipos e tipagens serão utilizadas na seção 4.8.5 que formaliza as noções de tipo e tipagem principal.

4.6 Operações de Supremo

Nesta seção serão apresentadas as definições de funções que calculam o supremo de tipos simples e substituições de acordo com as ordens parciais definidas anteriormente. Estas funções serão utilizadas para permitir a definição opcional de classes de tipos, quando o conjunto de instâncias for conhecido *a priori*.

4.6.1 Cálculo do Supremo de Tipos Simples

A função que calcula o supremo de dois tipos simples é chamada lcg (menor generalização comum[2]) e sua definição é apresentada na figura 4.7. A função lcg utiliza a função auxiliar gen, definida na figura 4.8. Esta função é usada para definir a variável que representará, no tipo resultado da operação, dois subtermos correspondentes e não unificáveis dos tipos a partir dos quais se está computando o supremo. Ela garante que uma mesma variável seja usada para representar os mesmos subtermos correspondentes. O parâmetro V representa o conjunto de variáveis ainda não utilizadas.

[2] do inglês: ***l**east **c**ommon **g**eneralization*.

$$
\begin{array}{rcl}
\mathrm{lcg}(\mu_1,\mu_2) & = & \mu \ \text{ onde } (\mu,S_1,S_2,V)=\mathrm{lcg}'(\mu_1,\mu_2,id,id,tv(\mu_1)\cup tv(\mu_2)) \\
\mathrm{lcg}'(T,\mu,S_1,S_2,V) & = & \begin{cases} (T,S_1,S_2,V) & \text{se } \mu = T \\ \mathrm{gen}(T,\mu,S_1,S_2,V) & \text{caso contrário} \end{cases} \\
\mathrm{lcg}'(\mu_1\,\mu_2,\mu_1'\,\mu_2',S_1,S_2,V) & = & \begin{cases} (\mu\,\mu',S,S',V_2) & \text{se } kind_{\mathbf{T}}(\mu_1)=kind_{\mathbf{T}}(\mu_2), \\ & (\mu,S_1',S_2',V_1)=\mathrm{lcg}'(\mu_1,\mu_1',S_1,S_2,V) \\ & (\mu',S,S',V_2)=\mathrm{lcg}'(\mu_2,\mu'^2,S_1',S_2',V_1) \\ \mathrm{gen}(\mu_1\,\mu_2,\mu_1'\,\mu_2',S_1,S_2,V) & \text{caso contrário} \end{cases} \\
\text{em todos os outros casos:} & & \\
\mathrm{lcg}'(\mu_1,\mu_2,S_1,S_2,V) & = & \mathrm{gen}(\mu_1,\mu_2,S_1,S_2,V)
\end{array}
$$

Figura 4.7. Definição da função lcg

$$
\mathrm{gen}(\mu_1,\mu_2,S_1,S_2,V)=\begin{cases} (\alpha,S_1,S_2,V) \text{ se } S_1(\alpha)=\mu_1, S_2(\alpha)=\mu_2 \text{ para algum } \alpha \\ (\alpha',S_1'\circ S_1,S_2'\circ S_2,V\cup\{\alpha'\}), \text{ caso contrário,} \\ \text{onde: } \alpha'\notin V, S_1'=[\alpha'\mapsto\mu_1], S_2'=[\alpha'\mapsto\mu_2] \end{cases}
$$

Figura 4.8. Definição dda função gen

4.6.2 Cálculo do Supremo de Substituições

O supremo de duas substituições é calculado pela função lcgs, definida como:

$$
\begin{array}{rcl}
\mathrm{lcgs}(S_1,S_2) & = & S, \text{ onde,} \\
V & = & tv(S_1)\cup tv(S_2) \\
(S,S',S'') & = & \mathrm{lcgs}'(\mathrm{dom}(S_1)\cup\mathrm{dom}(S_2),id,id,id) \\
\mathrm{lcgs}'(\emptyset,S,S_A,S_B) & = & S \\
\mathrm{lcgs}'(\alpha,S_A',S_B',V') & = & \mathrm{lcgs}'(S_1(\alpha),S_2(\alpha),S_A,S_B,V) \\
\mathrm{lcgs}'(\{\alpha\}\cup V,S,S_A,S_B) & = & \mathrm{lcgs}'(V,[\alpha\mapsto\mu]\circ S,S_A',S_B') \text{ se } V\neq\emptyset
\end{array}
$$

4.7 Satisfazibilidade de Restrições

O conjunto de restrições κ em um tipo $\sigma=\forall\overline{\alpha}.\kappa\Rightarrow\tau$ restringe o conjunto de tipos para os quais σ pode ser instanciado em um dado contexto Γ, conforme existam suposições em Γ que satisfaçam κ. Usaremos a notação $\Gamma\models^{sat}\kappa[S]$ para representar o fato que o conjunto de restrições κ é satisfeito em um contexto Γ pela substituição S. Definiremos a relação de satisfazibilidade gradualmente nesta seção, primeiramente para uma restrição e depois para conjuntos de restrições.

4.7.1 Satisfazibilidade de uma Restrição

A restrição $C\,\overline{\mu}$ representa um requerimento sobre um contexto. Sua presença no tipo de uma expressão exige a definição de uma instância da classe C que a satisfaça. Dizemos que uma instância $\forall\overline{\alpha}.\kappa \Rightarrow C\overline{\mu_1}$ satistfaz a restrição $C\,\overline{\mu}$ se as seguintes condições são verdadeiras:

1. Existe uma substituição S tal que $S(C\,\overline{\mu}) = S(C\,\overline{\mu_1})$.
2. Existe uma substituição S' que satifaz o conjunto de restrições κ, introduzido pela instância $\forall\overline{\alpha}.\kappa \Rightarrow C\overline{\mu_1}$.

Sendo as duas condições anteriores verdadeiras, temos que substituição

$$S_0 = (S' \circ S)|_{tv(C\,\overline{\mu})}$$

satisfaz a restrição $C\,\overline{\mu}$ no contexto Γ. A operação $S_1|_V$ representa, informalmente, a restrição do domínio de uma substituição S_1 sobre um conjunto de variáveis V. A definição formal desta operação é:

$$S_1|_V = \{[\alpha \mapsto \tau] \mid \alpha \in dom(S_1) \cap V \wedge [\alpha \mapsto \tau] \in S_1\}$$

A regra que define a satisfazibilidade de uma restrição é a regra (SAT_1), definida como:

$$\frac{\kappa \Rightarrow C\,\overline{\mu_1} \in \Gamma^{ins}(C) \qquad S(C\,\overline{\mu}) = S(C\,\overline{\mu_1}) \qquad \Gamma \models^{sat} \kappa[S'] \quad S_0 = S' \circ S|_{tv(C\,\overline{\mu})}}{\Gamma \models^{sat} \{C\,\overline{\mu}\}[S_0]}\ (\mathrm{SAT}_1)$$

Figura 4.9. Regra para Satisfazibilidade de uma Restrição

4.7.2 Satisfazibilidade de um Conjunto de Restrições

A satisfazibilidade de um conjunto de restrições $\kappa = \{C\,\overline{\mu_1}, ..., C\,\overline{\mu_n}\}$ é definida de maneira análoga a satisfazibilidade de uma restrição. Dizemos que um conjunto κ de restrições é satisfazível em um contexto Γ se existe uma substituição S tal que S seja uma solução para todos os elementos de κ.

Desta maneira podemos definir indutivamente a satisfazibilidade de um conjunto de restrições. Temos que um conjunto vazio de restrições é trivialmente satisfazível por qualquer substituição. Consideraremos que um conjunto $\kappa = \emptyset$ é satisfeito pela substituição identidade (id), o que nos permite definir a regra $\mathrm{SAT}_\emptyset$ como:

Todo conjunto de restrições κ, com $|\kappa| \geq 1$ pode ser escrito da seguinte maneira:

$$\frac{}{\Gamma \models^{sat} \emptyset\,[id]}\,(\mathrm{SAT}_{\emptyset})$$

Figura 4.10. Regra para Satisfazibilidade de um Conjunto Vazio de Restrições

$$\kappa = \{C\,\overline{\mu}\} \cup \kappa',$$

onde $C\,\overline{\mu} \notin \kappa'$. Tal fato, permite-nos deduzir a regra de satisfazibilidade para um conjunto não vazio de restrições (SAT_{many}):

$$\frac{\Gamma \models^{sat} \{C\,\overline{\mu}\}[S_1] \qquad \Gamma \models^{sat} \kappa[S_2]}{\Gamma \models^{sat} \{C\,\overline{\mu}\} \cup \kappa[S_2 \circ S_1]}\,(\mathrm{SAT}_{many})$$

Figura 4.11. Satisfazibilidade de um Conjunto não Vazio de Restrições

Intuitivamente, a solução para um conjunto κ com $n \geq 1$ restrições é obtida realizando a composição das substituições que satisfazem cada uma das restrições $\delta \in \kappa$, o que é representado pela regra SAT_{many} na figura 4.11.

O problema de satisfazibilidade de um conjunto de restrições κ em um contexto Γ é indecidível [Volpano & Smith, 1991]. Em [Camarão et al., 2004] é apresentado um algoritmo incompleto para o problema que tem sido utilizado com sucesso na implementação de um algoritmo de inferência de tipos baseado no Sistema CT [Camarao et al., 2007, Camarao & Figueiredo, 1999a].

O algoritmo de satisfazibilidade utilizado neste trabalho é baseado na proposta de [Camarão et al., 2004], porém as condições que deverão ser impostas para que se garanta a terminação da satisfazibilidade ainda deverão ser definidas. Na versão atual do protótipo implementado, adotamos as restrições impostas por [Sulzmann et al., 2006a].

4.8 Definição do Sistema de Tipos

Nesta seção é apresentado um sistema de tipos para Haskell com suporte a classes com múltiplos parâmetros sem a necessidade de dependências funcionais ou famílias de tipos [Camarão et al., 2009].

Antes de apresentarmos as regras do sistema de tipos, definiremos alguns conceitos necessários. Primeiramente, definiremos a operação de Fechamento de Conjunto de Restrições, que será utilizada na definição de tipos bem formados (seção 4.8.2). A seção 4.8.3 define a simplificação e igualdade de tipos.

4.8.1 Fechamento de Conjunto de Restrições

O fechamento de conjunto de restrições κ com respeito a um conjunto de variáveis de tipo V, $\kappa|_V^*$, é definido como:

$$\kappa|_V = \{C\ \overline{\mu} \in \kappa \mid tv(\overline{\mu}) \cap V \neq \emptyset\}$$
$$\kappa|_V^* = \begin{cases} \kappa|_V & \text{se } tv(\kappa|_V) \subseteq V \\ \kappa|_{tv(\kappa|_V)}^* & \text{caso contrário} \end{cases}$$

Figura 4.12. Fechamento de um Conjunto de Restrições

Intuitivamente, $\kappa|_V^*$ calcula o conjunto de restrições que possui variáveis de tipo *alcançáveis* a partir do conjunto de variáveis V. Uma variável de tipo α é alcançável se $\alpha \in V$ ou $\alpha \in tv(\delta)$, para algum $\delta \in \kappa$ que possua alguma variável $\beta \in V$ ou β está em outra restrição que possui uma variável presente no conjunto V. Dizemos que uma variável γ é *inalcançável* se $\gamma \in tv(\kappa) - \kappa|_V^*$.

Para exemplificar esta definição, considere o seguinte conjunto de restrições:

$$\kappa = \{\texttt{F a b}, \texttt{G a c}\}$$

Temos que $\{\texttt{F a b}, \texttt{G a c}\}|_{\{c\}}^* = \{\texttt{F a b, G a c}\}$. A restrição `G a c` está presente no fechamento pois $tv(\texttt{G a c}) \cap \{c\} \neq \emptyset$. Já a restrição `F a b` está presente no fechamento porquê possui uma variável de tipo (`a`) que está presente em uma restrição que possui uma variável (`c`) pertencente ao conjunto {`c`}.

4.8.2 Tipos Bem Formados

Um tipo restringido $\kappa \Rightarrow \tau$ é bem formado em um contexto Γ, $\Gamma \models \kappa \Rightarrow \tau$, se $\Gamma \models^{wf} \kappa \Rightarrow \tau[s]$ é provável de acordo com a definição da figura 4.13:

$$\boxed{\Gamma \models^{wf} \kappa \Rightarrow \tau\ [S]}$$

$$\frac{\begin{array}{l}\text{se existe uma } \textit{única} \text{ substituição } S \text{ tal que } \Gamma \models^{sat} \kappa_0\ [S] \ \text{ e } \ tv(S\kappa_0) = \emptyset \\ \text{once } \ \kappa_0 = (\kappa - \kappa|_V^*) \ \cup \ \{\delta \in \kappa \mid tv(\delta) = \emptyset\} \\ \qquad V = tv(\tau) - tv(\Gamma)\end{array}}{\Gamma \models^{wf} \kappa \Rightarrow \tau\ [S]}$$

Figura 4.13. Definição de Tipo Bem Formado.

Informalmente, um tipo $\kappa \Rightarrow \tau$ é bem formado se o conjunto de todas as restrições em κ que não possuem variáveis de tipo ou que possuem variáveis "inalcançáveis" possui

uma única solução em Γ. É importante observar que nesta definição consideramos que o contexto Γ possui apenas restrições de classes e instâncias bem formadas de acordo com a definição presente na seção 4.4.2

4.8.3 Simplificação e Igualdade de Tipos

Seja $\kappa \Rightarrow \tau$ um tipo restringido tal que $\Gamma \models^{wf} \kappa \Rightarrow \tau\ [S]$ é provável para algum S, então:

- $S\,\tau = \tau$
- $S\,\kappa$ não contém variáveis inalcançáveis, i.e. $S\kappa = (S\kappa)|^*_V$, onde $V = tv(\tau) - tv(\Gamma)$.

Isto ocorre porquê a substituição S é a única solução para κ em Γ. Desta maneira, temos que o tipo $\kappa \Rightarrow \tau$ pode ser especializado para $\kappa' \Rightarrow \tau$, onde:

$$\kappa' = S\,\kappa - \{\delta \in S\kappa \,|\, tv(\delta) = \emptyset\}.$$

Dá-se o nome de *simplificação* à remoção de restrições satisfeitas (uma restrição δ é satisfeita se $tv(\delta) = \emptyset$) em κ. Usaremos a notação $\Gamma \models \kappa \Rightarrow \tau \gg \kappa' \Rightarrow \tau$, para denotar a simplificação de κ para κ'.

A operação de simplificação pode ser estendida para tipos. Um tipo σ pode ser simplificado para σ' — $\Gamma \models \sigma \gg \sigma'$, onde $\sigma = \forall\overline{\alpha}.\kappa \Rightarrow \tau$ e $\sigma' = \forall\overline{\alpha}'.\kappa' \Rightarrow \tau$ — se $\Gamma \models \kappa \Rightarrow \tau \gg \kappa' \Rightarrow \tau$ e $\overline{\alpha}' = tv(\kappa' \Rightarrow \tau) \cap \overline{\alpha}$.

A simplificação de tipos produz tipos iguais em Γ. Denotamos por $\Gamma \models \sigma \equiv \sigma'$ a relação de equivalência entre tipos que consiste do fecho reflexivo, simétrico e transitivo da relação de simplificação de tipos.

4.8.4 O Sistema de Tipos

Um sistema de tipos dirigido por sintaxe[3] para *core-Haskell* é apresentado na figura 4.14. As regras são similares as regras do sistema de tipos de Hindley-Milner [Milner, 1978], exceto pelo uso da instanciação de tipos restringidos (seção 4.5.3.1) na regra (VAR). Na regra (LET) é requerido que um tipo restringido seja bem formado ($\Gamma \models \kappa \Rightarrow \tau$), antes que este seja incluído em Γ e na regra (APP) o tipo da aplicação $e\,e'$ inclui as restrições que ocorrem nos tipos de e e e'.

Cabe ressaltar que o sistema de tipos apresentado na figura 4.14 provê suporte a classes de tipos com múltiplos parâmetros, mas ainda não permite a declaração opcional de classes de tipos.

[3] do inglês: *Syntax Directed Type System*

$$\boxed{\Gamma \vdash e : \kappa \Rightarrow \tau}$$

$$\frac{x : \sigma \in \Gamma \quad \Gamma \models \sigma \leq_{\Sigma} \kappa \Rightarrow \tau}{\Gamma \vdash x : \kappa \Rightarrow \tau} \quad \text{(VAR)}$$

$$\frac{\Gamma, x : \tau' \vdash e : \kappa \Rightarrow \tau}{\Gamma \vdash \lambda x.\, e : \kappa \Rightarrow \tau' \to \tau} \quad \text{(ABS)}$$

$$\frac{\Gamma \vdash e : \kappa \Rightarrow \tau' \to \tau \quad \Gamma \vdash e' : \kappa' \Rightarrow \tau'}{\Gamma \vdash e\, e' : \kappa \cup \kappa' \Rightarrow \tau} \quad \text{(APP)}$$

$$\frac{\Gamma \vdash e : \kappa \Rightarrow \tau \quad \Gamma \models \kappa \Rightarrow \tau \quad \Gamma, x : \sigma \vdash e' : \kappa' \Rightarrow \tau'}{\Gamma \vdash \texttt{let}\ \ x = e\ \ \texttt{in}\ \ e' : \kappa' \Rightarrow \tau'} \quad \text{(LET)}$$

$$\text{onde:} \quad \sigma = \forall \overline{\alpha}.\kappa \Rightarrow \tau \qquad \overline{\alpha} = tv(\kappa \Rightarrow \tau) - tv(\Gamma)$$

Figura 4.14. Regras do Sistema de Tipos

Na próxima seção serão apresentadas propriedades importantes do sistema de tipos descrito nesta seção.

4.8.5 Propriedades do Sistema de Tipos

O sistema de tipos possui a importante propriedade de que a derivabilidade de tipos bem formados é fechada sobre a aplicação de substituições (se todas as restrições de classe e instâncias são bem formadas). Essa propriedade é conhecida como o *lema da substituição*[4] [Mitchell, 1996].

> **Lema 1** (Substituição): *Seja* $\Gamma \vdash e : \kappa \Rightarrow \tau$. *Então* $\Gamma \models \kappa \Rightarrow \tau$ *e para todo* S *tal que* $S\Gamma \models S(\kappa \Rightarrow \tau)$, *nós temos que* $S\Gamma \vdash e : S(\kappa \Rightarrow \tau)$.

É comum dizer que o sistema de tipos "tem a propriedade de tipo principal" se, para qualquer par formado por um contexto de tipos e uma expressão da linguagem, ou não existe tipo derivável para a expressão neste contexto ou existe um tipo principal para a expressão neste contexto [Damas & Milner, 1982, Faxén, 2003]. O conceito de tipo principal não deve ser confundido com o de *tipagem principal* [Wells, 2002, Jim, 1996]. Informalmente, temos:

[4] do inglês: *Substitution lemma*

Tipo Principal

Dado:	um termo e e um contexto Γ.
Existe:	um tipo σ que representa todos os tipos possíveis de e em Γ.

Tipagem Principal

Dado:	um termo e.
Existe:	um tipo σ e um contexto Γ tais que Γ contém as suposições "mínimas" para tipagem de e, e σ representa todos os tipos que podem ser derivados para e em Γ.

As definições a seguir formalizam os conceitos de tipo e tipagem principal.

Definição 1 (Tipo Principal). O tipo principal de uma expressão e em um contexto Γ é o menor limite superior da ordem parcial $\sigma \leq_\Sigma \sigma'$, definida na seção 4.5.3.1.

Definição 2 (Tipagem Principal). A tipagem principal de uma expressão e em um contexto mais externo Γ_o é o menor limite superior da ordem parcial (seção 4.5.5.1) $(\sigma_1, \Gamma_1) \leq_\omega (\sigma_2, \Gamma_2)$ de todas as tipagens (σ, Γ) tal que $\Gamma \vdash e : \kappa \Rightarrow \tau$ é provável e $\sigma = \forall \overline{\alpha}.\kappa \Rightarrow \tau$, onde $\overline{\alpha} = tv(\kappa \Rightarrow \tau) - tv(\Gamma)$.

4.9 Inferência de Tipos

O algoritmo para inferência de tipos é apresentado na figura 4.15 como um sistema de provas dirigidas por sintaxe[5] de julgamentos $\Gamma \vdash_{\mathtt{I}} e : (\kappa \Rightarrow \tau, \Gamma')$.

Seja $\Theta(\Gamma)$ o contexto de tipos mais externo contendo restrições de classes, instâncias e o tipo principal de cada símbolo sobrecarregado, isto é:

$$\Theta(\Gamma) = \bigcup_C \Gamma^{cls}(C) \cup \bigcup_C \Gamma^{ins}(C) \cup \bigcup_x \Gamma(x)$$

onde C representa um nome de classe e x um símbolo sobrecarregado. Note que $\Theta(\Gamma)$ não é igual a Γ_o (definido na seção 4.4), uma vez que $\Theta(\Gamma)$ inclui somente informações sobre símbolos sobrecarregados e Γ_o inclui toda a informação de símbolos sobrecarregados ou não.

Adicionalmente, o algoritmo de inferência utiliza as seguintes notações:

[5] do inglês: *Syntax-directed Proof System.*

$$\boxed{\Gamma \vdash_{\mathtt{I}} e : (\kappa \Rightarrow \tau,\, \Gamma')}$$

$$\frac{\Gamma(x) = \forall\, \overline{\alpha}.\, \kappa \Rightarrow \tau}{\Gamma \vdash_{\mathtt{I}} x : \big(\kappa \Rightarrow \tau,\, \{x : \Gamma^*(x)\}\big)} \qquad (VAR_{\mathtt{I}})$$

$$\frac{\Gamma \vdash_{\mathtt{I}} (e : \kappa \Rightarrow \tau, \Gamma')}{\Gamma \ominus x \vdash_{\mathtt{I}} \lambda x.\, e : (\kappa \Rightarrow \tau' \to \tau,\, \Gamma' \ominus x)} \qquad (ABS_{\mathtt{I}})$$

$$\text{where: } \tau' = \begin{cases} \tau & \text{if } x : \tau \in \Gamma' \\ \alpha & \text{otherwise, } \alpha \text{ fresh} \end{cases}$$

$$\frac{\Gamma \vdash_{\mathtt{I}} e : (\kappa \Rightarrow \tau, \Gamma_1) \qquad \Gamma \vdash_{\mathtt{I}} e' : (\kappa' \Rightarrow \tau', \Gamma_2)}{\Gamma \vdash_{\mathtt{I}} e\, e' : S'S(\kappa \cup \kappa') \Rightarrow S\alpha,\; S\Gamma_1 \cup S\Gamma_2)} \qquad (APP_{\mathtt{I}})$$

$$\begin{aligned} \text{where:}\quad & S = unify(\{\tau = \tau' \to \alpha\} \cup st(\Gamma_1, \Gamma_2)) \\ & S' = wf(S(\kappa \cup \kappa' \Rightarrow \alpha), \Gamma),\ \alpha \text{ fresh} \end{aligned}$$

$$\frac{\Gamma \vdash_{\mathtt{I}} e : (\kappa \Rightarrow \tau, \Gamma_1) \qquad \Gamma, \{x : \sigma\} \vdash_{\mathtt{I}} e' : (\kappa' \Rightarrow \tau', \Gamma_2)}{\Gamma \vdash_{\mathtt{I}} \text{let}\;\; x = e \;\; \text{in} \;\; e' : (S'S\kappa' \Rightarrow S\tau',\; S\Gamma' \ominus x)} \qquad (LET_{\mathtt{I}})$$

$$\begin{aligned} \text{where:}\quad & S_0 = wf(\kappa \Rightarrow \tau, \Gamma_1) \\ & \sigma = \forall\, \overline{\alpha}.\, S_0\kappa \Rightarrow \tau \\ & \overline{\alpha} = tv(S_0\kappa \Rightarrow \tau) - tv(\Gamma_1) \\ & S = unify(st(\Gamma_1, \Gamma_2)) \\ & S' = wf(S(\kappa \cup \kappa' \Rightarrow \tau'), \Gamma) \end{aligned}$$

Figura 4.15. Algoritmo para Inferência de Tipagens Principais

$$\Gamma(x) = \begin{cases} \kappa \Rightarrow \tau & \text{se } x \text{ é um símbolo sobrecarregado e } \Gamma^{cls}(x) = \forall\, \overline{\alpha}.\, \kappa \Rightarrow \tau \\ & \text{ou } x \text{ é uma variável de tipo let ou lambda or e } x : \forall\, \overline{\alpha}.\, \kappa \Rightarrow \tau \in \Gamma \\ & \quad \text{onde as variáveis de tipo de } \kappa \Rightarrow \tau \text{ são novas variáveis de tipo.} \\ \alpha & \text{caso contrário, onde } \alpha \text{ é uma nova variável de tipo} \end{cases}$$

$\Gamma^*(x) = \Gamma(x) \cup \Theta(\Gamma)$
$\Gamma \ominus x = \Gamma - \{x : \sigma \mid x \text{ é uma variável de tipo let ou lambda}, \sigma \in \Gamma(x)\}$

Também são usadas as funções lb, st, $unify$ e wf, onde $unify$ é uma função para unificar dois tipos; $lb(\Gamma)$ denota o subconjunto de suposicções de tipo para variáveis de tipo lambda em Γ e $st(\Gamma, \Gamma')$ produz o conjunto de igualdades entre tipos simples

de cada variável lambda presente em em Γ e Γ' (que devem ser unificadas durante a inferência, já que estão restritas a possuir tipos monomórficos):

$$st(\Gamma, \Gamma') = \{\tau = \tau' \mid x : \tau \in lb(\Gamma),\ x : \tau' \in lb(\Gamma')\}$$

A função wf é utilizada para verificar se um tipo $\kappa \Rightarrow \tau$ é bem formado em um contexto Γ, de acordo com a definição de $\Gamma \models^{wf} \kappa \Rightarrow \tau$ apresentada na seção 4.8.2. A definição da função wf é apresentada na figura 4.16.

$$\begin{array}{l}
wf(\kappa \Rightarrow \tau, \Gamma) = \\
\quad let\ \ \kappa_0 = \kappa - \kappa|^*_{tv(\tau)} \cup \{\delta \in \kappa \mid tv(\delta) = \emptyset\} \\
\quad in\ if\ \kappa_0 = \emptyset\ then\ id \\
\qquad else\ case\ sat(\kappa_0, \Gamma)\ of \\
\qquad\qquad (Unsat, _) \quad \rightarrow\ "error:\quad unsatisfiability \ldots " \\
\qquad\qquad (Single, S) \quad \rightarrow\ if\ tv(S\kappa_0) = \emptyset\ then\ S \\
\qquad\qquad\qquad\qquad\quad\ \ else\ "error:\quad ambiguity \ldots " \\
\qquad\qquad (Many, _) \quad \rightarrow\ "error:\quad ambiguity \ldots "
\end{array}$$

Figura 4.16. Função para determinar se um tipo é bem formado

A definição de wf usa a função $sat(\kappa,\Gamma)$, definida em [Camarão et al., 2004, Camarao et al., 2007], que implementa a relação $\Gamma \models^{sat} \kappa[S]$ (seção 4.7) para obter uma solução para o problema de satisfazibilidade (κ, Γ), caso esta exista. Mais precisamente, a função sat um conjunto de possíveis soluções para o problema. Caso não exista solução ($Unsat$) ou exista mais de uma ($Many$) uma mensagem de erro é retornada indicando a insatisfazibilidade ou ambiguidade de κ em Γ. Se existir uma única solução para κ esta é utilizada para especializar este conjunto de restrições. Note que o teste de satisfazibilidade é executado em wf apenas quando existem variáveis inalcançáveis (seção 4.8.1) em κ, e que portanto precisam ser resolvidas ou o tipo em questão será considerado ambíguo.

4.10 Conclusão

Neste capítulo foi apresentado um sistema de tipos para Haskell que permite a definição de classes de tipos com múltiplos parâmetros que não requer extensões como dependências funcionais e famílias de tipos para evitar a ocorrência de tipos ambíguos. Tal sistema é uma solução minimalista para a a adoção de classes com múltiplos parâme-

tros em Haskell [Camarão et al., 2009], já que não exige nenhuma extensão ao sistema de classes de tipos.

Capítulo 5

Planejamento das Atividades

Este capítulo apresenta as atividades a serem realizadas para a elaboração desta tese. As seguintes atividades já foram realizadas:

1. Definição formal de um sistema de tipos para Haskell que dê suporte a classes de tipos com múltiplos parâmetros sem a necessidade de extensões.
2. Implementação de um *front-end* para Haskell que utilize o sistema de tipos proposto.

As seguintes atividades devem ser realizadas para alcançar os objetivos traçados nesta tese:

1. Modificação do sistema de tipos já elaborado para permitir a definição opcional de classes de tipos. Para isso, deve-se modificar o sistema de tipos para acomodar a utilização de operações de supremo, definidas na seção 4.7.
2. Modificação do protótipo para permitir a definição opcional de classes de tipos.
3. Avaliação do protótipo para inferência de tipos de bibliotecas Haskell que utilizam dependências funcionais ou famílias de tipos.
4. Elaboração de restrições sobre definições de classes e instâncias para garantir a terminação do algoritmo para satisfazibilidade de restrições.
5. Prova de corretude do algoritmo de inferência de tipos com relação ao sistema de tipos proposto.
6. Prova que o sistema de tipos proposto possui as propriedades de tipo e tipagem principal.

Até o presente momento o seguinte artigo foi publicado:

- *A Solution to Haskell's Multi-Parameter Type Class Dilemma.* Este artigo apresenta o sistema de tipos descrito no capítulo 4 deste trabalho. Publicado no XIII Simpósio Brasileiro de Linguagens de Programação em 2009.

Os seguintes artigos ainda serão elaborados com os resultados futuros deste trabalho:

- *Optional and Multi-Parameter Type Classes for Haskell.* Apresentação da versão final do sistema de tipos e algoritmo de inferência proposto neste trabalho.

- *Haskell and Principal Typings.* Trabalho que descreverá a prova formal da propriedade de tipagem principal para o sistema de tipos proposto.

A figura 5.1 apresenta o sumário proposto para esta tese. As seções a serem elaboradas estão em destaque.

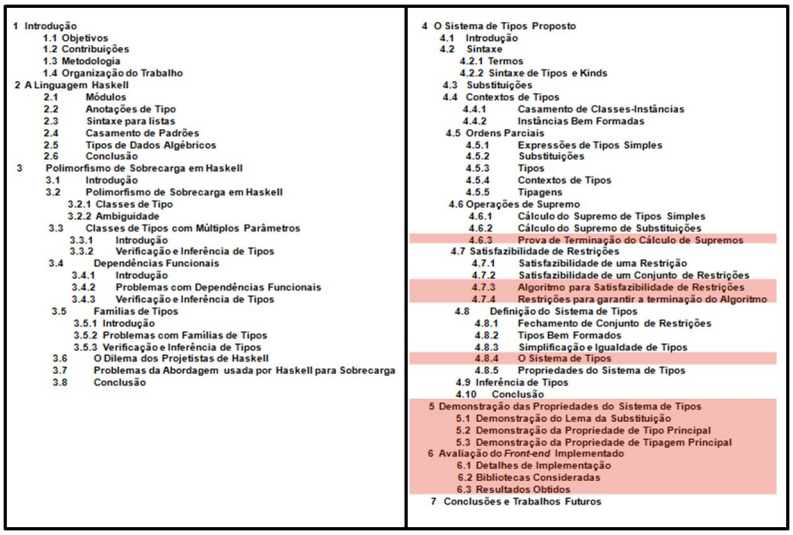

Figura 5.1. Sumário da Tese

O cronograma das atividades a serem realizadas é apresentado na figura 5.2.

Atividade	2010				
	Ago	Set	Out	Nov	Dez
Elaboração do Sistema de tipos para Classes de tipos opcionais					
Restrições para garatir a terminação do algoritmo de satisfazibilidade					

Atividade	2011											
	Jan	Fev	Mar	Abr	Mai	Jun	Jul	Ago	Set	Out	Nov	Dez
Modificação do protótipo front-end para permitir a definição opcional de classes de tipos												
Avaliação do *front-end* implementado												
Elaboração do Artigo 2												
Prova de Corretude do Algoritmo de inferência												
Prova de Propriedades do Sistema de tipos												
Elaboração do Artigo 3												
Conclusão e revisão do texto												

	2012
Atividade	Jan
Defesa da Tese	

Figura 5.2. Cronograma das Atividades

Referências Bibliográficas

[Augustsson, 1998] Augustsson, L. (1998). Cayenne—a language with dependent types. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pp. 239--250, New York, NY, USA. ACM.

[Baader & Nipkow, 1998] Baader, F. & Nipkow, T. (1998). *Term rewriting and all that*. Cambridge University Press, New York, NY, USA.

[Camarão et al., 2004] Camarão, C.; Figueiredo, L. & Vasconcellos, C. (2004). Constraint-set satisfiability for overloading. *Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming - PPDP '04*, pp. 67--77.

[Camarão et al., 2009] Camarão, C.; Ribeiro, R.; Figueiredo, L. & Vasconcellos, C. (2009). A Solution to Haskells Multi Parameter Type Class Dilemma. *Proceedings of the Brazilian Symposium on Programming Languages*.

[Camarao & Figueiredo, 1999a] Camarao, C. & Figueiredo, L. (1999a). Type inference for overloading. In *FLOPS'99 - International Workshop on Logic and Functional Programming*.

[Camarao & Figueiredo, 1999b] Camarao, C. & Figueiredo, L. (1999b). Type inference for overloading without restrictions, declarations or annotations. In *Fuji International Symposium on Functional and Logic Programming*, pp. 37–52.

[Camarao et al., 2007] Camarao, C.; Vasconcellos, C.; Figueiredo, L. & Nicola, J. (2007). Open and closed worlds for overloading: a definition and support for coexistence. *Journal of Universal Computer Science*, 13(6):874--890.

[Chakravarty et al., 2005a] Chakravarty, M. M. T.; Keller, G. & Jones, S. P. (2005a). Associated type synonyms. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pp. 241--253, New York, NY, USA. ACM.

[Chakravarty et al., 2005b] Chakravarty, M. M. T.; Keller, G.; Jones, S. P. & Marlow, S. (2005b). Associated types with class. *ACM SIGPLAN Notices*, 40(1):1--13.

[Chen & Xi, 2005] Chen, C. & Xi, H. (2005). Combining programming with theorem proving. In *In ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pp. 66--77. ACM Press.

[Chen et al., 1992] Chen, K.; Hudak, P. & Odersky, M. (1992). Parametric type classes. In *LFP '92: Proceedings of the 1992 ACM conference on LISP and functional programming*, pp. 170--181, New York, NY, USA. ACM.

[Damas & Milner, 1982] Damas, L. & Milner, R. (1982). Principal Type-Schemes for Functional Programs. pp. 207--212.

[Davey & Priestly, 1990] Davey, B. A. & Priestly, H. A. (1990). *Introduction to Lattices and Order*. Cambridge University Press.

[Duggan & Ophel, 2002] Duggan, D. & Ophel, J. (2002). Type-checking multi-parameter type classes. *J. Funct. Program.*, 12(2):133--158.

[Faxén, 2003] Faxén, K.-F. (2003). Haskell and principal types. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pp. 88--97, New York, NY, USA. ACM.

[Frühwirth, 1995] Frühwirth, T. (1995). Constraint handling rules. In *Constraint Programming: Basics and Trends, LNCS 910*, pp. 90--107. Springer-Verlag.

[Hudak et al., 2007] Hudak, P.; Hughes, J.; Jones, S. P. & Wadler, P. (2007). A history of haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pp. 12–1--edoformat12–55, New York, NY, USA. ACM.

[Jim, 1996] Jim, T. (1996). What are principal typings and what are they good for? In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 42--53, New York, NY, USA. ACM.

[Jones, 2000] Jones, M. P. (2000). Type Classes with Functional Dependencies. *Proceedings of the 9th European Symposium on Programming Languages and Systems*, (March).

[Jones & Diatchki, 2008] Jones, M. P. & Diatchki, I. S. (2008). Language and program design for functional dependencies. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pp. 87--98, New York, NY, USA. ACM.

[Jones & Diatchki, 2009] Jones, M. P. & Diatchki, I. S. (2009). Language and program design for functional dependencies. *ACM SIGPLAN Notices*, 44(2):87.

[Jones et al., 1997] Jones, S.; Jones, M. & Meijer, E. (1997). Type classes: an exploration of the design space.

[Jones, 2002] Jones, S. P., editor (2002). *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/.

[Milner, 1978] Milner, R. (1978). A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, (17):348--375.

[Mitchell, 1996] Mitchell, J. C. (1996). *Foundations of programming languages*. MIT Press, Cambridge, MA, USA.

[Peyton Jones et al., 2006] Peyton Jones, S.; Vytiniotis, D.; Weirich, S. & Washburn, G. (2006). Simple unification-based type inference for gadts. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pp. 50--61, New York, NY, USA. ACM.

[S. P. Jones and others, 1998] S. P. Jones and others (1998). GHC — The Glasgow Haskell Compiler website. http://www.haskell.org/ghc/.

[Schrijvers et al., 2008] Schrijvers, T.; Jones, P. & Others (2008). Type checking with open type functions. *ACM SIGPLAN Notices*, 43(9).

[Strachey, 2000] Strachey, C. (2000). Fundamental Concepts in Programming Languages. *High-Order and Symbolic Computation*, 13(1-2):11--49.

[Sulzmann et al., 2007] Sulzmann, M.; Chakravarty, M. M. T.; Jones, S. P. & Donnelly, K. (2007). System f with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pp. 53--66, New York, NY, USA. ACM.

[Sulzmann et al., 2006a] Sulzmann, M.; Duck, G.; Peyton-Jones, S. & Stuckey, P. (2006a). Understanding functional dependencies via constraint handling rules.

[Sulzmann et al., 2006b] Sulzmann, M.; Schrijvers, T. & Stuckey, P. J. (2006b). Principal type inference for ghcstyle multi-parameter type classes. In *In Proc. of APLAS'06*.

[Sulzmann et al., 2006c] Sulzmann, M.; Wazny, J. & Stuckey, P. J. (2006c). A framework for extended algebraic data types. In *In Proc. of FLOPS'06, volume 3945 of LNCS*, pp. 47--64. Springer-Verlag.

[Volpano, 1994] Volpano, D. M. (1994). Haskell-style overloading is np-hard. In *In Proceedings of the 1994 International Conference on Computer Languages*, pp. 88--94.

[Volpano & Smith, 1991] Volpano, D. M. & Smith, G. (1991). On the complexity of ml typability with overloading. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pp. 15--28, London, UK. Springer-Verlag.

[Wadler & Blott, 1989] Wadler, P. & Blott, S. (1989). How to make ad-hoc polymorphism less ad hoc. *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 60--76.

[Watt, 1990] Watt, D. A. (1990). *Programming language concepts and paradigms.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Wells, 2002] Wells, J. B. (2002). The essence of principal typings. In *In Proc. 29th Int'l Coll. Automata, Languages, and Programming, volume 2380 of LNCS*, pp. 913--925. Springer-Verlag.

[Wolfgang Jeltsch, 1998] Wolfgang Jeltsch (1998). Mensagem Enviada para a lista GHC users. http://www.haskell.org/pipermail/glasgow-haskell-users/2008-September/015562.html.