CLASSES DE TIPOS COM MÚLTIPLOS PARÂMETROS E OPCIONAIS EM HASKELL

RODRIGO GERALDO RIBEIRO

CLASSES DE TIPOS COM MÚLTIPLOS PARÂMETROS E OPCIONAIS EM HASKELL

Tese apresentada 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

Julho de 2013

Ribeiro, Rodrigo Geraldo

Multi-Parameter and Optional Type Classes in Haskell / Rodrigo Geraldo Ribeiro. — Belo Horizonte - MG, 2013
xxi, 121 f. : il. ; 29cm

Tese (doutorado) — Universidade Federal de Minas Gerais
Orientador: Carlos Camarão de Figueiredo

Linguagem de Programação, Haskell

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Classes de tipos com múltiplos parâmetros e opcionais em haskell

RODRIGO GERALDO RIBEIRO

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. CARLOS CAMARÃO DE FIGUEIREDO - Orientador
Departamento de Ciência da Computação - UFMG

PROF. ALBERTO PARDO COSTA
Universidad de la Republica

PROF. FERNANDO MAGNO QUINTÃO PEREIRA
Departamento de Ciência da Computação - UFMG

PROFA. LUCÍLIA CAMARÃO DE FIGUEIREDO
Departamento de Computação - UFOP

PROF. MARTIN SULZMANN
Informatik und wirtschaftinformatik - HsKA

PROF. ROBERTO DA SILVA BIGONHA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 30 de julho de 2013.

À minha querida esposa Dáfani e minha filha Teresa.

Agradecimentos

Agradeço ao Departamento de Ciência da Computação da Universidade Federal de Minas Gerais pela oportunidade de participar do seu programa de doutorado.

Agradeço ao meu orientador, professor e amigo Carlos Camarão pela atenção, incentivo, disponibilidade e paciência diante de minha inexperiência e nas muitas dificuldades que encontrei para concluir este trabalho.

À minha co-orientadora e amiga Lucília Figueiredo, que me acompanha desde os primeiros períodos de minha graduação na UFOP, pelos preciosos ensinamentos que foram essenciais em minha formação.

Aos professores do Departamento de Ciência da Computação pelas disciplinas que tive a oportunidade de cursar.

Aos meus amigos e colegas de trabalho do Departamento de Ciências Exatas e Aplicadas da UFOP, em especial, Elton e Leonardo pelos diversos cafés e bate-papos.

Gostaria de agradecer a meus pais pelo seu apoio em minha graduação. Sem tal apoio não teria a oportunidade de concluir meus estudos em Ciência da Computação.

Gostaria de agradecer a minha sogra Betânia por todo o apoio.

Gostaria de agradecer também minha amada esposa Dáfani por sua paciência, carinho e compreensão que estiveram presentes na longa jornada que culmina com a conclusão desta tese. Faltam-me palavras para descrever o quanto sou grato a ela por tudo. Sem o apoio de minha esposa posso afirmar categoricamente que este trabalho não seria concluído.

Finalmente, gostaria de dedicar este trabalho a minha recém chegada filha Teresa, que para mim é uma fonte de alegria inesgotável.

"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 — Humano, Demasiado Humano)

Resumo

A introdução de classes de tipos com múltiplos parâmetros em Haskell tem sido dificultada devido a problemas associados a ambiguidade, que ocorrem devido a uma falta de especialização durante a inferência de tipos. Este trabalho apresenta 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. Além disso, revemos o conceito de ambiguidade de Haskell de maneira a adotar a definição usual baseada em derivações de tipos. É permitida também a declaração de símbolos sobrecarregados sem a imposição de que seja definida uma classe de tipos para esses. Além disso, é desenvolvido e implementado um algoritmo de inferência de tipos correto e completo com respeito ao sistema de tipos definido.

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. Haskell's ambiguity definition is redefined as the usual definition based on type systems derivations. The definition of overloaded symbols without the need of specifying a type class is also allowed. A type inference algorithm that is sound and complete with respect to the proposed type system is presented and implemented.

Lista de Figuras

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 de programas. 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* ou *polimorfismo de Damas-Milner* 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 [Mitchell, 1996]. Esse tipo é usualmente chamado de tipo principal. Isso se deve ao fato de que o sistema de tipos permite deduzir vários tipos (infinitos, se o tipo possui variáveis de tipos e apenas um, caso contrário) para cada expressão. No entanto, como veremos no Capítulo 3, no caso de linguagens que estendem o suporte a polimorfismo paramétrico para permitir sobrecarga como em Haskell, isso introduz problemas de incoerência em definições da semântica da linguagem por indução em derivações do sistema de tipos. Neste trabalho usamos um sistema de tipos que permite a derivação de um único tipo para cada expressão. Isso será explicado detalhadamente nos Capítulos 3 e 4.

Diversas funções presentes em bibliotecas para manipulação de estruturas de dados são exemplos de funções que podem ser caracterizadas por polimorfismo paramétrico: 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 desse tipo de funções, chamadas de polimórficas.

O tipo 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 sobre valores de qualquer tipo que é instância de um determinado tipo, 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), analisadores sintáticos, conversão de valores para strings, 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 simplesmente de 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 $\forall\alpha.\,\texttt{String} \to \alpha$. Um sistema de tipos que adote uma política dependente do contexto permite a resolução de sobrecarga em expressões como: λ`x.read x == "abc"`. Neste exemplo, o tipo de `read` é determinado como sendo `String` $\to$ `String`. Isso ocorre porque `"abc"` em Haskell possui o tipo `String`, e o resultado de `read x` deve ter, então, o tipo `String`, uma vez que `(==)` tem tipo $\forall\,\alpha.\,\alpha \to \alpha \to$ `Bool`, e portanto os dois argumentos de `(==)`, `read x` e `"abc"`, devem ter o mesmo tipo (α). O fato de que o resultado de `read x` deve ter tipo `String`, por estar sendo usado na expressão (no contexto) `read x == "abc"`, caracteriza a sobrecarga dependente de contexto (a sobrecarga de `read` é resolvida, como tendo o tipo `String` $\to$ `String`, pelo contexto em que ocorre). O tipo de λ`x.read x == "abc"` é então `String` $\to$ `Bool`.

A linguagem *Haskell* permite combinar o *polimorfismo paramétrico* com o suporte à sobrecarga dependente de contexto. 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. Implementações de símbolos sobrecarregados são feitas em declarações de instâncias. Em uma declaração de instância são fornecidas as implementações para os nomes especificados em uma classe, com tipos que devem ser instâncias do tipo especificado na classe.

Na atual definição da linguagem [Jones, 2002], são permitidas classes com apenas um parâmetro. Classes com mais de um parâmetro não foram introduzidas na definição da linguagem devido a dificuldades existentes no tratamento de expressões ambíguas[1] que podem surgir devido ao uso de símbolos sobrecarregados. Os atuais compiladores e 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. No entanto, atualmente ainda não existe um algoritmo de inferência de tipos, definido e aceito pela comunidade, que incorpora o uso de dependências funcionais, apenas uma especificação em alto nível desse mecanismo [Jones, 2000, Jones & Diatchki, 2009]. Além disso, em algumas situações dependências funcionais não podem ser utilizadas, uma vez que pode não existir uma dependência funcional entre os parâmetros de uma classe. Por último, o uso de dependências funcionais estabelece um critério para ativação de testes de satisfazibilidade de restrições, que não é dependente apenas da existência de variáveis de tipo inalcançáveis no conjunto de restrições de um tipo. Analisaremos isso mais detalhadamente no Capítulo 3.

O dilema atual enfrentado pelos projetistas de Haskell é que classes com múltiplos parâmetros são muito úteis e devem ser introduzidas na linguagem, mas não há consenso na comunidade sobre como solucionar os problemas de ambiguidade e especialização de tipos com restrições [Committee, 2012].

1.1 Objetivos

O objetivo principal deste trabalho é a elaboração de um sistema e um algoritmo de inferência de tipos para Haskell que dê suporte a classes de tipos com múltiplos parâ-

[1] 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].

metros, sem a necessidade de extensões como dependências funcionais. O sistema de tipos e o algoritmo de inferência permitem também 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, foi desenvolvido um *front-end* de um compilador Haskell que implementa o algoritmo de inferência apresentado. São demonstradas as propriedades de correção e completude do algoritmo de inferência em relação ao sistema de tipos.

1.2 Contribuições

1. Formalização de um sistema de tipos para Haskell, com suporte a definição e uso de classes de tipos com múltiplos parâmetros, sem necessidade de mecanismos adicionais na linguagem, e que permite também a definição opcional de classes de tipos.

2. Formalização de um algoritmo de inferência de tipos correto e completo em relação ao sistema de tipos apresentado. Para isso, revemos o conceito de ambiguidade de Haskell de maneira a adotar a definição usual de ambiguidade baseada em derivações de tipos.

3. Implementação da fase de análise de um compilador ou interpretador para Haskell, que implementa o algoritmo de inferência apresentado e que permite 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 tipos com múltiplos parâmetros, implementada em compiladores como o GHC [S. P. Jones and others, 1998].

4. Formalização, em um assistente de provas de um algoritmo para calcular a generalização mínima de dois tipos. Este algoritmo é utilizado para calcular o tipo de funções sobrecarregadas sem a necessidade de declaração, pelo programador, de uma classe de tipos.

1.3 Metodologia

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

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

2. Definição de um sistema de tipos e de um algoritmo de inferência de tipos que permita a declaração opcional de classes de tipos. Implementação desse algoritmo no programa acima citado.

3. Demonstração de corretude e completude do algoritmo de inferência em relação ao sistema de tipos.

1.4 Organização do Trabalho

Além deste capítulo introdutório, este trabalho é dividido em duas 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 os Capítulos 4 e 5. O Capítulo 4 apresenta a definição formal do sistema de tipos elaborado, suas propriedades e descreve a implementação do algoritmo de inferência, baseado nesse sistema de tipos. O Capítulo 5 apresenta a abordagem usada para definição opcional de classes de tipos.

Capítulo 2

A Linguagem Haskell

Este capítulo apresenta uma breve introdução à linguagem Haskell. A abordagem utilizada na linguagem para suporte ao polimorfismo de sobrecarga é apresentada no Capítulo 3, que também apresenta e discute a noção de ambiguidade.

> "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 introdução à linguagem, considere o trecho de programa mostrado na Figura 2.1. Divideremos este capítulo em seções, onde cada seção aborda uma característica da linguagem.

2.1 Módulos

Programas em Haskell são compostos por uma sequência de *módulos*. Módulos provêem uma forma de 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:

```
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 = x : 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 = (fst (remove s xs), x : snd (remove s xs))
```

Figura 2.1. Um Módulo em Haskell

declarações de classes, instâncias, tipos de dados e declarações de valores, incluindo 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 não funcional `empty` e as funções `insert`, `member`, `search`, `remove` e `update` para manipulação de tabelas.

2.2 Anotações de Tipo

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

Todos os nomes definidos no módulo `Table` são *polimórficos*. Por exemplo, a constante `empty` tem tipo `Table a`, que é sinônimo do tipo `[(String,a)]`. Isso indica que `empty` pode ser utilizado em contextos que requeiram valores de tipos que são instâncias do tipo ∀`a. [(String,a)]`, como por exemplo `[(String,Bool)]`, `[(String,Int)]`, ∀`a. [(String,[a])]` 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` → `Table a` → `a`, que especifica que esta função recebe como parâmetro um valor do tipo `String` e retorna uma função, que recebe uma lista de pares compostos por um valor de tipo `String` e um elemento de um tipo qualquer e retorna como resultado um elemento deste tipo. Em geral dizemos, informalmente, que `search` recebe dois parâmetros (um de cada "vez"), um valor de tipo `String` e uma lista de pares.

Cabe ressaltar que anotações de tipos são, em geral, opcionais em programas Haskell, uma vez que o compilador é capaz de inferir o tipo para cada expressão. Este processo de determinar o tipo de expressões é chamado de *inferência de tipo*. Caso o programador forneça uma anotação de tipo para uma expressão, o compilador verifica se a definição 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 isso, existe em Haskell uma sintaxe especial para representar esse 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` → `[a]` → `[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:

```
[True, False]
```

é uma abreviação para

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

No módulo `Table`, a função `member` usa outra 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:

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

O último tipo de *açúcar sintático* disponível na linguagem Haskell para listas é apresentado suscintamente a seguir, por meio de exemplos:

- `['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 Definições de Funções

Funções em Haskell podem ser definidas de modo a explorar o casamento de padrões e o uso de guardas. Estes dois recursos para definições de funções são explicados nas próximas sub-seções.

2.4.1 Casamento de Padrões

Um padrão é uma construção sintática que pode envolver o uso de constantes e variáveis, introduzida para definir o mecanismo de casamento de padrões. O casamento de padrões é uma operação usada na passagem de parâmetros. Basicamente, consiste simplesmente em que uma constante só casa com si própria, e uma variável casa com qualquer expressão. O casamento de uma variável provoca uma associação da variável à expressão com a qual houve o casamento.

O *casamento de padrões* desempenha um papel importante nas definições de funções em linguagens funcionais modernas. 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 utiliza o padrão `[]` e a segunda equação utiliza o padrão `(x:xs)`. O padrão

`(:)x xs` é um exemplo de uma constante funcional `(:)` aplicada à variável `x` e à variável `xs`.

2.4.2 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 avaliação da guarda (expressão booleana) especificada na definição resulta em um valor verdadeiro.

2.5 Tipos de Dados Algébricos

Nas Figuras 2.2 e 2.3 apresentadas a seguir são mostradas declarações de um tipo de dados algébrico e de uma função que recebe valores desse tipo como argumento. O objetivo é ilustrar características básicas da definição e o 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: a palavra reservada `data` é usada na declaração de `Maybe`. A declaração introduz `Maybe` como um *construtor de tipos* que possui dois *construtores de dados*: `Nothing` e `Just`. O tipo `Maybe a` é polimórfico, ou seja, quantificado universalmente sobre uma ou mais variáveis de tipo. Para cada tipo `t` instanciado, ou seja, substituído pela variável de tipo `a` em `Maybe a`, existe um novo tipo de dados, `Maybe t`. 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 de tipo `Maybe t`, 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` indica que um valor deste tipo pode ser uma folha (`Leaf`), cujo tipo corresponde a um produto trivial, de apenas um tipo, ou um nodo

construído com o construtor `Node`, cujo tipo corresponde a um produto de um tipo `a` com dois tipos `Tree a` (que correspondem às sub-árvores esquerda e direita).

```
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 breve introdução à linguagem Haskell e algumas de suas características básicas: sistema de módulos, anotações de tipos, açúcar sintático para listas, casamento de padrões e definições de tipos de dados algébricos. Para cada uma dessas características foram apresentados exemplos ilustrativos de sua utilização. O próximo capítulo aborda um dos recursos principais da linguagem — a possibilidade de definir símbolos sobrecarregados usando uma política de resolução de sobrecarga dependente de contexto.

Capítulo 3

Sobrecarga Dependente de Contexto em Haskell

Este capítulo apresenta uma breve introdução à abordagem adotada na linguagem Haskell para sobrecarga. Apresenta também abordagens apresentadas na literatura para suporte a classes com múltiplos parâmetros em Haskell, e discute a noção de ambiguidade.

3.1 Introdução

Strachey foi o primeiro a utilizar o termo *polimorfismo ad-hoc* para se referir a funções que podem ser aplicadas a argumentos de diferentes tipos, mas que se comportam de acordo com o tipo do argumento para o qual são aplicadas [Strachey, 2000]. Isso é o que chamamos de sobrecarga independente de contexto. Neste texto, será usado o termo *polimorfismo de sobrecarga* para denominar não só esse tipo de polimorfismo mas também o polimorfismo introduzido pelo mecanismo de sobrecarga dependente de contexto usado em Haskell (veja Capítulo 1). Linguagens que provêem suporte ao polimorfismo de sobrecarga permitem ao programador fazer várias definições de funções, todas com o mesmo nome. A tarefa de determinar qual função é chamada pode ser realizada pelo compilador, que toma essa decisão com base em informações do contexto onde o nome da função é usado.

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 grande 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 [Camarão & Figueiredo, 1999].

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

Desde sua versão original baseada no trabalho de Wadler e Blot [Wadler & Blott, 1989], várias extensões foram propostas para o sistema de classes de tipos de Haskell. Em sua maioria, essas extensões tinham o intuito de permitir a utilização de classes de tipos com múltiplos parâmetros. Dentre estas podemos citar: Classes de Tipos Paramétricas [Chen et al., 1992], Dependências Funcionais [Jones, 2000, Jones & Diatchki, 2009, Sulzmann et al., 2006a] e Famílias de Tipos[1] [Schrijvers et al., 2008, Chakravarty et al., 2005b, Chakravarty et al., 2005a, Kiselyov et al., 2010].

3.2 Sobrecarga Dependente de Contexto 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.

A função `primEqInt` é uma função pré-definida, de tipo `Int` $\rightarrow$ `Int` $\rightarrow$ `Bool`, que verifica a igualdade de dois números inteiros. Considerando as duas instâncias definidas

[1] do inglês: *Type families*.

```
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)
   _ == _ = False
```

Figura 3.1. Exemplo de classe de tipos e instâncias

para a classe `Eq`, tem-se que as 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 tem uma restrição da pela classe `Eq`.

A função `member`, definida na Figura 3.2, tem um tipo denotado em Haskell por `Eq a ⇒ a → [a] → Bool`. Temos que `Eq a` é uma restrição que limita os tipos para os quais a variável `a` pode ser instanciada, aos tipos que são instâncias da classe `Eq`.

```
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.

Classes de tipos podem ser declaradas de maneira a formar hierarquias. Na Figura 3.3, a classe `Ord` é definida como *subclasse* de `Eq`. Isso indica que, para que um tipo seja instância da classe `Ord`, ele deve ser também instância da 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 `Ord a ⇒ a → [a] → Bool`. Caso a classe `Ord` não fosse definida como subclasse de `Eq` esse tipo seria

`(Ord a, Eq a) ⇒ a → [a] → Bool`.

Uma característica interessante de funções sobrecarregadas definidas por classes de tipo é que estas podem ser traduzidas em funções não sobrecarregadas equivalentes, que recebem um argumento extra, denominado dicionário [Wadler & Blott, 1989]. Um dicionário é um registro que contém, 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 outro 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 tipo

de dicionários da 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 `(==) e (/=)`.

Para cada uma das funções definidas na classe, é gerada uma função de projeção, que seleciona a função correspondente no dicionário.

A tradução da função `member` (Figura 3.5), definida na Figura 3.2, possui um parâmetro extra, correspondente ao dicionário da classe `Eq`, que representa a restrição `Eq a` presente em seu tipo. Além disso, a utilização do símbolo sobrecarregado `(==)` é substituída pela expressão `eq d`, onde `eq` é uma função de projeção e `d` um dicionário:

```
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.

A linguagem Haskell permite também, na definição de classes, prover implementações *padrão* (*default*) para os nomes membros de uma classe. Na definição da classe `Eq` (Figura 3.1), temos definições padrão para `(==)` e `(/=)` como sendo:

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

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

3.3 Ambiguidade

Nesta seção discutimos a noção de ambiguidade usando sua definição convencional (Seção 3.3.1) e na Seção 3.3.2 apresentamos a definição utilizada pela linguagem Haskell. Finalmente, na Seção 3.3.3, analisamos a abordagem para ambiguidade utilizada pela linguagem Haskell.

3.3.1 Definição de Ambiguidade

Expressões que possuem tipos ambíguos são rejeitadas por compiladores e interpretadores de Haskell, por não ser possível definir para estas um significado preciso.

Usualmente, define-se a semântica de expressões por indução sobre derivações de tipo [Mitchell, 1996], mas sistemas de tipos para Haskell apresentados na literatura [Jones, 1995a, Vytiniotis et al., 2011] permitem que sejam construídas duas ou mais derivações para o mesmo tipo de uma expressão, resultando em um problema conhecido como não coerência. Dizemos que uma semântica é coerente se esta atribui o mesmo significado para uma expressão independentemente da maneira como o tipo desta foi derivado. Mais formalmente:

> "Sejam Δ e Δ' derivações de $\Gamma \vdash e : \sigma$[2] e $\Gamma' \vdash e : \sigma$, respectivamente, tais que Γ e Γ' atribuem o mesmo tipo para qualquer variável livre x presente em e. Se a semântica atribuída á expressão e utilizando-se as derivações Δ e Δ' for igual, isto é: $[\![\Gamma \vdash e : \sigma]\!]_\eta = [\![\Gamma' \vdash e : \sigma]\!]_\eta$, dizemos que a semântica em questão é coerente". [Mitchell, 1996]

Dizemos que uma expressão e de tipo σ é ambígua, se existem duas derivações distintas Δ e Δ' que atribuem à expressão e dois diferentes valores semânticos. Exemplos são discutidos nas Seções 3.3.2 e 3.3.3.

3.3.2 Ambiguidade em Haskell

Um problema da abordagem atualmente utilizada em Haskell para o polimorfismo de sobrecarga é a definição e o tratamento de *ambiguidade*, como ilustrado a seguir.

Exemplo 1. 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)
```

A função `show` converte um valor, de qualquer tipo definido como instância da classe `Show`, para um valor de tipo `String`, enquanto `read` faz o inverso para qualquer tipo que é instância da classe `Read`. Intuitivamente, `f` deveria se comportar como a função identidade sobre valores do tipo `String`, já que primeiramente converte o valor `s` (de tipo `String`) em um valor de um tipo que é instância da classe `Read` e na sequência converte o resultado de `read s` para uma `String` utilizando a função `show`. Porém, na

[2] A notação $\Gamma \vdash e : \sigma$ especifica que á expressão e possui tipo σ em um contexto Γ derivável utilizando um conjunto de regras de dedução que descreve o sistema de tipos para a linguagem em questão. Maiores detalhes sobre sistemas de tipos são apresentados no Capítulo 4.

definição de `f` não existe maneira de determinar como instanciar a variável de tipo `a` presente no tipo de `read`. Desta maneira, temos que o tipo inferido para `f` é:

$$\texttt{f :: (Read a, Show a)} \Rightarrow \texttt{String} \rightarrow \texttt{String}$$

As regras adotadas na definição da linguagem Haskell não examinam o contexto para instanciação de variáveis, ou seja, a variável de tipo `a` presente nas restrições `Read a` e `Show a` não é instanciada, portanto, a sobrecarga presente na definição de `f` não é resolvida.

Expressões cujo tipo possui restrições contendo variáveis que não podem ser instanciadas posteriormente são ditas ambíguas em Haskell e são rejeitadas pelo compilador ou interpretador. Um tipo polimórfico tem, em Haskell, a forma $\forall\,\overline{\alpha}.\ P \Rightarrow \tau$, onde $\overline{\alpha}$ é uma sequência de variáveis de tipo e P são restrições de classes de tipo. Haskell define que um tipo $\forall\,\overline{\alpha}.P \Rightarrow \tau$ é ambíguo se existe alguma variável de tipo presente nas restrições (P) que não está presente no tipo (τ). Diz-se que tal ocorrência, neste tipo, é ambígua, como ocorre no caso da variável `a` do tipo de `f` (Exemplo 1).

Uma maneira de evitar esse erro em Haskell é utilizar expressões com anotações de tipo. Por exemplo, a definição de `f` pode ser feita usando uma anotação do tipo de `read s`, como feito a seguir:

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

Com essa anotação de tipo, a definição de `f` deixa de ser ambígua, pois nesse contexto a sobrecarga é resolvida, de modo a usar a instância de `read` com tipo `String` $\rightarrow$ `Int` e, consequentemente, a sobrecarga de `Show` é também resolvida, de modo a usar a instância de `show` para o tipo `Int` $\rightarrow$ `String`.

Ambiguidades em operações da classe `Num`[3] são comuns. Para evitar a necessidade de usar uma anotação de tipo em toda subexpressão numérica, Haskell adota uma regra bastante *ad hoc*, explicada a seguir, para permitir a eliminação de 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 ambiguidade, se (cf. [Jones, 2002]):

- `v` aparece somente em restrições da forma `C v`, onde `C` é uma classe, e

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

- 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 da classe `Num` 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 ambiguidade.

Para prover suporte a classes com múltiplos parâmetros, o compilador Haskell GHC utiliza uma condição diferente para considerar que um determinado tipo é ou não ambíguo. Um tipo $\forall\,\overline{\alpha}.\ P \Rightarrow \tau$ é considerado ambíguo se P possui alguma variável inalcançável a partir de τ, ou se alguma restrição em P não menciona nenhuma variável em $\overline{\alpha}$. Dizemos que uma variável α, presente em um conjunto de restrições P, é alcançável (e inalcançável, caso contrário) se: 1) α está presente em τ ou 2) α está presente em alguma restrição de P que possua variáveis alcançáveis[4] [S. P. Jones and others, 2012]. Com esta regra, o tipo $\forall$`a.C a b` $\Rightarrow$ `a` é aceito, enquanto $\forall$`a b.Eq b` $\Rightarrow$ `Int` não, pois a restrição `Eq b` não menciona a variável quantificada `a`.

3.3.3 Análise sobre a abordagem de Haskell para ambiguidade

Conforme apresentado na Seção 3.3.2, Haskell utiliza uma condição sintática para determinar quando um tipo é ambíguo ou não. Porém, conforme apresentaremos no exemplo a seguir, esta caracterização de ambiguidade é inadequada, pois exclui diversos programas que não são ambíguos, isto é, que possuem uma única tradução que pode ser obtida através da derivação de seu tipo.

Exemplo 2. Considere o seguinte trecho de código Haskell:

```
class Show a where show ::  a → String
class Read a where read ::  String → a
instance Show Bool where ...
instance Read Bool where ...
instance Read Int where ...

f x = show (read x)
```

[4] Até onde sabemos, o primeiro trabalho a definir ambiguidade em função da existência de variáveis inalcançáveis foi [Camarão & Figueiredo, 1999].

O tipo inferido para `f` é `(Show a, Read a)` $\Rightarrow$ `String` $\rightarrow$ `String`, que é considerado ambíguo pelo compilador GHC, pois a variável de tipo `a` presente nas restrições é inalcançável.

Observando as definições de classes e instâncias presentes no contexto da definição de `f`, podemos observar que somente as instâncias `Show Bool` e `Read Bool` podem ser utilizadas para a derivação de um tipo para `f`. Isso nos permite concluir que esta expressão não é ambígua (de acordo com a definição de ambiguidade apresentada na Seção 3.3.1), pois existe uma única derivação de tipo para `f`, a que utiliza as instâncias `Show Bool` e `Read Bool`.

O fato do tipo de uma expressão possuir variáveis inalcançáveis não necessariamente implica em ambiguidade, mas sim que a resolução de qual implementação deve ser utilizada para os símbolos sobrecarregados desta expressão deve ser neste ponto realizada. Em Haskell, a existência de variáveis inalcançáveis é utilizada para caracterizar ambiguidade, o que entra em conflito com a definição usual baseada na existência de derivações distintas para o tipo de uma expressão.

Uma expressão só pode ser considerada ambígua se houver duas ou mais maneiras de resolver a sobrecarga desta, quando a resolução de sobrecarga não puder ser deferida, isto é, quando o tipo da expressão possuir variáveis inalcançáveis e houver mais de um maneira de instanciar estas variáveis de forma a determinar qual implementação deve ser utilizada para os nomes sobrecarregados em questão.

No Capítulo 4 discutiremos com detalhes esta alternativa, que permite a definição de uma semântica coerente para Haskell.

3.4 Classes de Tipos com Múltiplos Parâmetros

3.4.1 Introdução

Classes de tipos com múltiplos parâmetros possuem diversas aplicações práticas [Jones et al., 1997, Jones, 2000, 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
```

Esse exemplo foi usado em [Jones, 2000]. A classe de tipos `Collects a b` define operações sobre coleções e seus tipos. A variável `a` representa o tipo dos elementos e `b` o tipo 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 essa 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 ambiguidade adotada em Haskell (apresentada na Seção 3.3). Essa função é considerada ambígua em Haskell já que em seu tipo (`empty::(Collects a b) ⇒ b`) a variável `a` presente nas restrições não ocorre no tipo simples (`b`).

Uma solução para esse 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 de essa declaração não apresentar problemas de ambiguidade em relação ao símbolo `empty`, ela só pode ser instanciada para coleções formadas por um construtor de tipos `c` aplicado a um tipo `a`. Para resolver estes problemas, que ocorrem devido á definição de ambiguidade usada em Haskell em conjunto com o uso de classes de múltiplos parâmetros, diversas propostas foram elaboradas. A próxima seção apresenta duas destas propostas.

3.4.2 Trabalhos Anteriores para Verificação e Inferência de Tipos

Nesta seção apresentamos alguns trabalhos existentes na literatura para verificação e inferência de tipos na presença de classes de tipos com múltiplos parâmetros. É sabido que, mesmo diante da restrição de que classes de tipos devem possuir apenas

um parâmetro, o problema de satisfazibilidade de restrições para tipos polimórficos é $\mathcal{NP}$-difícil. Sem restrições sobre a quantidade de parâmetros de uma classe e sem restrições sobre a satisfazibilidade de restrições em tipos, este problema torna-se indecidível [Volpano, 1994].

São descritas a seguir duas abordagens para verificação de tipos em programas que utilizem classes de tipos com múltiplos parâmetros em Haskell, sem a necessidade de extensões como dependências funcionais (seção 3.5) e famílias de tipos (seção 3.6). A abordagem apresentada neste trabalho é o tema no Capítulo 4.

3.4.2.1 A Abordagem de Duggan e Ophel

Duggan e Ophel apresentam um sistema de tipos e algoritmo de inferência [Duggan & Ophel, 2002] nos quais são feitas restrições que assemelham-se à utilização de dependências funcionais [Jones, 2000]. Para isso, toda classe de tipos deve ser da forma:

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

É requerido que as instanciações dos parâmetros α_1, ..., α_m devem unicamente determinar as instanciações dos parâmetros β_1, ..., β_n, para m e n declarados pelo programador. 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 instanciação de α_2 como `Matrix` α_1. Para contornar este problema, [Duggan & Ophel, 2002] permitem o uso de restrições para impedir a instanciação de variáveis de tipos. Uma restrição como $\alpha_2 \neq$ `Matrix` α_3 significa que a variável α_2 não pode ser instanciada para um tipo que possua `Matrix` como seu construtor mais externo. Usando este tipo de restrição, o trecho de código anterior pode ser escrito como:

```
instance (Mult α2 β γ, α2 ≠ Matrix α3) ⇒
                Mult α2 (Matrix β) (Matrix γ) where...
```

Além disso, os autores propõem utilizar no algoritmo de inferência uma estratégia para resolução de sobrecarga que é denominada *resolução de sobrecarga baseada em*

unificação[5], definida da seguinte maneira. Dado um conjunto de restrições P, seja $P\downarrow = (S^r, P^r)$ um par onde S^r é uma substituição e P^r é um conjunto de restrições que não foram resolvidas após a aplicação de S^r a P, isto é: $P^r = \{\pi \mid \pi \in S^r\, P \wedge tv(\pi) \neq \emptyset\}$[6]. A partir da configuração inicial (id, P), obtemos $P\downarrow$ pela execução dos seguintes passos:

1. Seja (S, P) o estado atual do algoritmo, $C\,\overline{\mu} \in P$ uma restrição, $P_1 \Rightarrow C\,\overline{\mu_1}$ uma instância da classe C, $S' = unify(\overline{\mu}, \overline{\mu_1})$ uma substituição que satisfaça todas as restrições $\alpha_i \neq \mu_i \in P$ e $P' = \{\pi \mid \pi \in S\,P \wedge tv(\pi) \neq \emptyset\} \cup P_1$. Então, a configuração atual do algoritmo passa a ser $(S' \circ S,\ S'\,P')$.

2. Seja (S, P) o estado atual do algoritmo, $P_1 \Rightarrow C\,\overline{\mu_1}$ uma instância da classe C e $C\,\overline{\mu_1}\,, C\,\overline{\mu_2} \in P$ 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 $1 \leq i \leq n$ 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 $(S' \circ S,\ S'\,P')$ onde $P' = P - \{C\,\overline{\mu_1}\}$.

3. O algoritmo falha caso não exista uma instância que unifique com alguma restrição em P.

Para uma melhor compreensão deste algoritmo, consideraremos dois exemplos que exibem como cada uma das regras funciona.

Exemplo 3. 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)
```

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

[6]A notação $tv(\pi)$ denota o conjunto de variáveis livres presentes na restrição π. A definição formal desta função é apresentada no Capítulo 4.

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

$$\texttt{Eq a, Name Employee a}$$

Supondo que todas as definições do prelúdio de Haskell[7] 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:

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

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`.

Exemplo 4. 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 as seguintes restrições geradas para o tipo de `f`:

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

Pela regra 2, temos que $C\,\overline{\mu_1} = \texttt{Add a b c}$, $C\,\overline{\mu_2} = \texttt{Add a b d}$, 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:

$$\texttt{f :: Add a b c} \Rightarrow \texttt{a} \rightarrow \texttt{b} \rightarrow \texttt{(c,c)}$$

ao invés de

$$\texttt{f :: (Add a b c, Add a b d)} \Rightarrow \texttt{a} \rightarrow \texttt{b} \rightarrow \texttt{(c,d)}$$

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

O primeiro tipo apresentado para `f` é, em nossa visão, incorreto. O motivo é que esse tipo impossibilita que a função (+) seja utilizada de maneira polimórfica. Para ilustrar o problema em questão, considere o seguinte trecho de código que estende o trecho de código contendo a definição da função `f` e da classe `Add`.

```
instance Add Bool Bool Bool where ...
instance Add Bool Bool String where ...

g = let (i,j) = f False True in (not i, j ++ ‘‘!’’)
```

Caso o tipo inferido para a função `f` seja

$$\texttt{f :: Add a b c} \Rightarrow \texttt{a} \rightarrow \texttt{b} \rightarrow \texttt{(c,c)}$$

temos que a função `g` não pode ser tipada, já que o resultado de `f` é um par de valores de um mesmo tipo `c`. Porém, caso o tipo de `f` seja

$$\texttt{f :: (Add a b c, Add a b d)} \Rightarrow \texttt{a} \rightarrow \texttt{b} \rightarrow \texttt{(c,d)}$$

a função `g` pode ser tipada, já que o resultado de `f` é um par de valores de tipos possivelmente diferentes.

Outro problema da abordagem proposta por [Duggan & Ophel, 2002] 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 próximo exemplo ilustra o problema de não terminação.

Exemplo 5. O programa da Figura 3.6 é um exemplo que causa a não terminação do algoritmo de inferência.

```
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

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

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

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:

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

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

$$P' \ = \{\pi \mid \pi \in S\,P \wedge tv(\pi) \neq \emptyset\} \cup P_1$$

onde $P_1 = \{\texttt{Foo a b}\}$ e $S = id$. Desta maneira, temos que P' será igual a:

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

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

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

Com isso temos que o estado do algoritmo será descrito pelo par $(S' \circ id,\, P')$ 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` $\Rightarrow$ `Foo [a] [b]` produzindo a substituição $S'' = id$ e o seguinte conjunto de restrições:

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

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

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

3.4.2.2 A Abordagem de Sulzmann

Outra alternativa para verificação e inferência de tipos na presença de classes de tipos com múltiplos parâmetros é apresentada em [Sulzmann et al., 2006b]. Nesse trabalho,

os autores estabelecem as seguintes restrições para a decidibilidade do problema de inferência:

1. O contexto presente em uma declaração de instância pode referenciar somente varáveis de tipos e, em cada restrição desse contexto, todas as variáveis nela contidas devem ser distintas.

2. Em uma declaração de instância `instance` $P \Rightarrow$ `C` $\overline{\mu}$, pelo menos um tipo μ_i, $1 \leq i \leq |\overline{\mu}|$, não deve ser uma variável, e devemos ter que $tv(P) \subseteq tv(\overline{\mu})$.

3. Instâncias não devem ser sobrepostas. Para quaisquer duas instâncias:

   `instance` $P_1 \Rightarrow$ `C` $\overline{\mu_1}$

   `instance` $P_2 \Rightarrow$ `C` $\overline{\mu_2}$

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

Se o programa obedecer essas condições, é provado pelos autores que, além de terminar para qualquer entrada, o processo de inferência produz tipos principais [Sulzmann et al., 2006b]. A prova utiliza uma transformação de um programa Haskell, contendo classes, instâncias e tipos de dados que satisfazem essas restrições, para um programa expresso por meio de *regras de manipulação de restrições*[8] [Frühwirth, 1995].

Um problema dessa abordagem é que as restrições impostas sobre declarações de instâncias são onerosas, impedindo a definição de alguns exemplos de interesse prático. O próximo exemplo apresenta uma definição de instância que não é permitida.

Exemplo 6. Considere a seguinte definição de um tipo de dados para representar uma mônada livre (exemplo retirado de [Voigtländer, 2008]):

```
data Free f a = Return a | Roll (f (Free f a))
```

Uma possível definição de uma instância da classe `Show` para `Free` seria:

```
instance (Show (f (Free f a)), Show a) => Show (Free f a) where ...
```

Tal instância viola a restrição 1, pois a restrição `Show (f (Free f a))` referencia não apenas variáveis de tipo. As restrições descritas em [Sulzmann et al., 2006b] foram implementadas em versões anteriores do compilador GHC, que hoje utiliza regras para aceitação de definições de instâncias mais flexíveis (mediante diretivas de compilação).

Na versão 7.4.2 do compilador GHC, este exemplo só é aceito utilizando-se a extensão `UndecidableInstances`, que desabilita todas as restrições sintáticas sobre definições de instâncias.

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

3.5 Dependências Funcionais

3.5.1 Introdução

A utilização de classes de tipo com múltiplos parâmetros, apesar de ser uma extensão útil para a definição de símbolos sobrecarregados, facilita a definição de expressões que possuem tipos ambíguos, de acordo com a definição de ambiguidade usada em Haskell. Como exemplo (cf. [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 ...
```

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 ambíguos, de acordo com a definição de ambiguidade usada em Haskell. 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 [Jones, 2002] 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
```

ocorrerá um erro de tipo, alertando que não existem instâncias que satisfaçam às 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 Int` não podem ser satisfeitas.

É possível contornar 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` poderia ser definida como:

```
Add a b r | a b → r where ...
```

A expressão `a b → r` significa que os tipos para os quais são instanciadas as variáveis `a` e `b` devem identificar para qual tipo a variável `r` deve ser instanciada. Dependências funcionais restringem as possíveis instâncias que um programador pode declarar para uma certa classe de tipos, do mesmo modo que classes de tipos são usadas para restringir os possíveis valores para os quais variáveis de tipo podem ser instanciadas em tipos polimórficos. Com isso, a dependência funcional `a b → r` na classe `Add` impede a definição das seguintes instâncias:

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

Isso ocorre porque os tipos correspondentes às variáveis `a` e `b` não determinam univocamente o tipo correspondente à variável `r`. Nestas duas instâncias, as variáveis `a` e `b` são instanciadas para o tipo `Int`, o que não determina o tipo para o qual a variável `r` pode ser instanciada, uma vez que `r` pode ser instanciada como `Int` ou `Float`. Para contornar este problema, uma destas instâncias tem que ser removida.

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 á não terminação do processo de inferência [Sulzmann et al., 2006a]. O próximo exemplo ilustra essa situação.

Exemplo 7. Considere o trecho de código apresentado na Figura 3.7.

```
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

Como `f` utiliza o símbolo sobrecarregado `(*)` em sua definição, o tipo de `f` possui a restrição:

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

Porém, a dependência `a b` $\rightarrow$ `c` e a instância:

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

fazem com que tenhamos 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.

Na Seção 3.5.2, são apresentadas restrições que podem ser usadas para garantir a corretude e terminação do algoritmo de inferência.

3.5.2 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.

3.5.2.1 Restrições para Garantir Corretude e Terminação

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

- Seja P um contexto presente em uma classe ou instância. Para cada restrição $C\,\overline{\mu} \in P$ temos que $\overline{\mu}$ deve ser formado apenas por variáveis de tipo e todas estas devem ser distintas.
- Em uma declaração de instância `instance` $P \Rightarrow$ `C` $\overline{\mu}$ `where...`, pelo menos um dos tipos $\mu_i \in \overline{\mu}$, $1 \leq i \leq |\overline{\mu}|$, não deve ser uma variável de tipo.

- Instâncias não devem ser sobrepostas. Isto é, para quaisquer duas instâncias:

  `instance` $P_1 \Rightarrow$ `C` $\overline{\mu_1}$ `where...`

  `instance` $P_2 \Rightarrow$ `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] introduz restrições sobre a definição de dependências funcionais:

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

  `class` $P \Rightarrow$ `C` $\overline{\alpha}$ `|` $fd_1, \ldots, fd_n$

  `instance` $P_1 \Rightarrow$ `C` $\overline{\mu_1}$ `where...`
  `instance` $P_2 \Rightarrow$ `C` $\overline{\mu_2}$ `where...`

  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:

  $$S\,\{\mu_{i1}, ..., \mu_{ik}\} = S\{\mu'_{i1}, ..., \mu'_{ik}\} \rightarrow S\,\mu_{i0} = S\,\mu'_{i0}$$

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

- **Cobertura**[9]: Considere a seguinte declaração da classe `C`, e uma instância qualquer dessa classe:

  `class` $P \Rightarrow$ `C` $\overline{\alpha}$ `|` $fd_1, \ldots, fd_n$

  `instance` $P_1 \Rightarrow$ `C` $\overline{\mu_1}$ `where...`

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

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

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

```
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 consistência é utilizada para evitar declarações de instâncias inconsistentes, como o exemplo de código da Figura 3.8 a seguir. 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`, o que viola a dependência funcional $a\ b \rightarrow c$ especificada na declaração da classe `Mult`.

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 trecho de código da Figura 3.9.

```
class Mult a b c | a b → c where
   (*) ::  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 os dois primeiros parâmetros da classe `Mult` sejam instanciados para `Int` e `Vector Int`. Tal instanciação não determina obrigatoriamente um valor para a variável `c`, já que $\{\texttt{c}\} \not\subseteq tv(\{\texttt{a, Vector b}\})$. A violação da condição de cobertura por uma declaração de instância acarreta 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, faz-se necessária a seguinte restrição:

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

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

[9] do inglês: *Coverage*.

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

$$\texttt{instance}\ P_1 \Rightarrow \texttt{C}\ \overline{\mu_1}\ \texttt{where}\ \ldots$$

deve ser verdade que $tv(P_1) \subseteq tv(\overline{\mu_1})$.

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].

3.5.2.2 Detecção de Ambiguidade

Como definido na Seção 3.3, uma expressão e é dita ser semanticamente ambígua se duas ou mais denotações distintas podem ser obtidas para ela, usando derivações distintas de um mesmo tipo para a expressão [Mitchell, 1996]. De acordo com a definição da linguagem Haskell, um tipo $\forall\overline{\alpha}.\, P \Rightarrow \tau$ é considerado ambíguo se $\exists v.v \in (tv(P) \cap \overline{\alpha}) \wedge v \notin tv(\tau)$ [Jones, 2002].

Para apresentar como é feita a detecção de ambiguidade utilizando dependências funcionais, primeiramente precisamos introduzir a seguinte definição. Seja F um conjunto de dependências funcionais e $J \subseteq I$ um conjunto de índices. O fechamento de J com respeito a F, J_F^+, é definido como 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 univocamente determinado pelos índices $i \in J$ ou pelas dependências $(X \rightarrow Y) \in F$. A partir desta definição podemos definir como é feita a detecção de ambigüidade utilizando dependências funcionais. O tipo $\forall\,\overline{\alpha}.\, P \Rightarrow \tau$ é considerado ambíguo se $\exists v.v \in (tv(P) \cap \overline{\alpha}) \wedge v \notin (tv(\tau))^+_{F_P}$, onde F_P é definido como: $F_P = \{tv(\overline{\mu}_X) \rightarrow tv(\overline{\mu}_Y) \mid (C\ \overline{\mu}) \in P \wedge (X \rightarrow Y) \in F_C\}$, $F_C = \{fd_1, ..., fd_n\}$ é o conjunto de dependências funcionais da classe `C` e $\overline{\mu}_X$ representa a projeção[10] da sequência $\overline{\mu}$ sobre X.

Exemplo 8. Considere o seguinte trecho de programa:

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

[10]Intuitivamente, a projeção de uma sequência sobre um conjunto $X \subseteq I$, em que I é o conjunto de índices que representam os parâmetros de uma classe, nada mais é que a sequência dos componentes de $\overline{\mu}$ correspondentes aos índices em X.

```
class C a b | a → b where
    c ::  a → b

h ::  (C a b, Eq b) ⇒ a → Bool
h x = (c x) == (c x)
```

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

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

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`:

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

Informalmente, o tipo

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

será ambíguo se existir uma variável que seja universalmente quantificada e pertença às restrições presentes em P e não esteja contida no fechamento das variáveis do tipo simples $\tau = \texttt{a} \rightarrow \texttt{Bool}$ 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:

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

e o fechamento deste sobre as variáveis de tipo de $\tau = \texttt{a} \rightarrow \texttt{Bool}$ é:

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

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

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

será ambíguo se:

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

mas:

- $P = \{\texttt{C a b, Eq b}\}$, e temos então que $tv(P) = \{\texttt{a,b}\}$.
- Como $\tau = \texttt{a} \rightarrow \texttt{Bool}$, temos que $tv(\tau) = \{\texttt{a}\}$ e que $\{\texttt{a}\}^{+}_{F_{\{\texttt{C a b, Eq b}\}}} = \{\texttt{a,b}\}$.

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

3.5.2.3 Especialização de Tipos

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

1. Suponha um tipo com 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 $\overline{\mu_2}_Y$.

2. Suponha que seja inferido um tipo com uma restrição $C\,\overline{\mu_1}$ e que exista uma instância:

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

   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].

Exemplo 9. Considere o seguinte trecho de programa:

```
class Mult a b c | a b → c where
  (.*.)  ::  a → b → c
instance Mult Matrix Matrix Matrix -- (1)
instance Mult Matrix Vector Matrix -- (2)

m1, m2, m3 ::  Matrix
-- definition of m1, m2 and m3

m = (m1 .*.  m2) .*.  m3
```

O tipo de `m` é:

```
(Mult Matrix Matrix a, Mult a Matrix b) ⇒ b
```

que pode ser especializado para `Matrix` utilizando a dependência funcional `a b → c` conforme a seguir. De acordo com a regra 2, primeiramente devemos obter a projeção

dos tipos da restrição `Mult Matrix Matrix a` com respeito ao domínio da dependência funcional na declaração da classe `Mult`. Essa projeção é dada por:

$$\{\texttt{Matrix,Matrix,a}\}_{\{a,b\}} = \{\texttt{Matrix,Matrix}\}$$

que é idêntica à projeção da declaração de instância (1). Logo, temos que os tipos correspondentes à projeção com respeito à imagem dessa dependência funcional devem ser iguais. Desta maneira, temos que a variável de tipo `a` é instanciada para o tipo `Matrix`, resolvendo a restrição `Mult Matrix Matrix a`. A especialização de `Mult Matrix Matrix b`[11] é feita de maneira similar.

3.6 Famílias de Tipos

3.6.1 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 e escrever programas completos no nível de tipos. Em Haskell, duas extensões ao sistema de tipos permitem expressar computações no nível de tipos[12]: dependências funcionais e famílias de tipos.

Famílias indexadas de tipos, ou simplesmente famílias de tipos, são uma extensão ao sistema de tipos de Haskell que permite 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 que um programa determine, em tempo de execução, quais são seus construtores de dados ao invés de fixá-los estaticamente. Como um primeiro exemplo, considere uma definição de uma família para representar mapeamentos finitos, análoga a uma definição de classe de tipo convencional, exceto pela presença de uma *família de tipos associada*: `data GMap k :: * → *`. Observe que esta declaração define um nome para esta família (`GMap`), uma variável de tipos (`k`) e o respectivo *kind*[13] de `GMap k`.

[11] Observe que essa restrição é obtida a partir de `Mult a Matrix b` instanciando a variável de tipo `a` para `Matrix`.

[12] do inglês: *Type level*

[13] *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.

Cabe ressaltar que a família `GMap` utiliza como parâmetro a mesma variável que é utilizada pela classe de tipos na qual esta 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 uma 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::* → * → *`[14] 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.

Na Figura 3.13 é 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 seus elementos é `Elem [e] = e`. Agora, considere a seguinte função `ins`:

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

[14] Se `GMap k` possui *kind* `* → *`, temos necessariamente que a variável `k` possui *kind* `*` e o construtor de tipos `GMap` possui, então, *kind* `* → * → *`.

```
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.

e o respectivo tipo inferido para ela:

```
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. Isso restringe as possíveis instâncias da família `Elem`, para que estas sejam iguais a `Char`, o que é representado no tipo de `ins` como `Elem c ∼ Char`. Tal restrição é denominada *restrição de igualdade de tipos* [Sulzmann et al., 2007, Schrijvers et al., 2008]. Restrições de igualdade, cuja forma geral é $t_1 \sim t_2$ — 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 desse novo tipo de restrições à linguagem.

Famílias são uma extensão proposta recentemente [Schrijvers et al., 2008], que visa oferecer as mesmas funcionalidades 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 $\sim$ representa a igualdade entre tipos, podemos substituir `F a` por `G (F a)` em `G (F a)`, resultando em `G (G (F a))`, o que pode levar a não terminação do algoritmo de satisfazibilidade.

Outro problema relacionado às restrições de igualdade é que estas ainda não estão totalmente 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 seguinte classe utilizando dependências funcionais:

```
class C a b | a → b where ...
```

Isao ocorre porque a restrição de igualdade (`F a` $\sim$ `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 especificada. 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` $\rightarrow$ `b`).

Na Seção 3.6.2 são apresentadas restrições impostas sobre a utilização de famílias de tipos para garantir a terminação do processo de inferência.

3.6.2 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 ao mecanismo de dependências funcionais para uso de classes com múltiplos parâmetros, e para desenvolvimento de programas em nível de tipos[15] e uso dos chamados tipos de dados algébricos generalizados[16]. Esta seção apresenta, informalmente, como famílias e restrições de igualdade de tipos são utilizadas durante o processo de verificação / inferência de tipos para Haskell.

3.6.2.1 Restrições para Definição de Famílias de Tipos

Na Seção 3.6.1 motivamos, de maneira breve, a necessidade de impor restrições sobre a definição de famílias de tipos para garantir a terminação do algoritmo de inferência. Em [Schrijvers et al., 2008], é definido como representar instâncias de famílias de tipos

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

como um sistema de reescrita de termos confluente e que termina para todas as entradas [Baader & Nipkow, 1998], desde que todas as instâncias atendam às seguintes restrições:

- As cabeças de instâncias de uma família não devem ser sobrepostas. Na declaração de uma instância, denomina-se *cabeça* a parte da definição que está à esquerda do símbolo "=" e *corpo* da instância a parte que está do lado direito. Por exemplo, considere a seguinte instância de uma família `F`:

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

  Nesse exemplo, a cabeça da declaração é `F [a] k` e o corpo é `(a,k)`.

- O corpo de uma instância de uma família de tipos não deve possuir aplicações de famílias aninhadas. Observe que com esta restrição, não é possível definir, por exemplo, a seguinte instância (onde `G` é uma família de tipos):

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

  Essa definição leva a não terminação do sistema de reescrita associado, uma vez que é gerada a igualdade `F a` $\sim$ `G (F a)`.

3.6.2.2 Inferência de Tipos

Considere o problema de realizar inferência de tipos para um programa (envolvendo famílias de tipos), o qual não possui qualquer 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. Por exemplo, considere a seguinte expressão:

```
λ c → (insert 'x' c, 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
```

Para inferir o tipo de λ `c` $\rightarrow$ `(insert 'x' c, 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 `'x'`) com o tipo do primeiro parâmetro desta função (`Elem` α), o que produz a restrição `Elem` $\alpha \sim$`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

$$\texttt{Elem}\ \alpha \sim \texttt{Char}$$

passa a ser igual a `Elem [`β`]` $\sim$ `Char`. Pela instância (definida na Figura 3.13):

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

temos que a restrição anterior pode ser simplificada para: $\beta \sim$`Char` e com isso, o tipo da expressão λ `c` $\rightarrow$ `(insert 'x' c, length c)` é inferido como:

$$\texttt{String} \rightarrow \texttt{(String, Int)}$$

já que o tipo de `insert` é especializado para `Char` $\rightarrow$ `String` $\rightarrow$ `String`, devido à restrição de igualdade `Elem` $\alpha \sim$ `Char` e à instância definida para a família `Elem`.

3.6.2.3 Verificação de Tipos

Segundo [Schrijvers et al., 2008] os maiores problemas relativos a utilização de famílias de tipos ocorrem na 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. Restrições de igualdade de tipos são introduzidas por anotações de tipos fornecidas pelo programador e por 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. O problema de verificação de tipos consiste então em determinar uma solução para um conjunto de restrições de igualdade de tipos. Como este problema não está relacionado diretamente a utilização de classes de tipos com múltiplos parâmetros, não daremos maiores detalhes, pois este problemas está fora do escopo deste trabalho.

3.7 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. Acredita-se que para isso é necessário adotar uma extensão que permita a utilização dessas classes [Committee, 2012]. Até o presente momento, dependências funcionais e famílias de tipos têm 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 essas 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 à 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.

3.8 Problemas da Abordagem Usada por Haskell Para Sobrecarga

Os principais problemas da abordagem usada por Haskell para sobrecarga são os seguintes:

- Em Haskell o conceito de ambiguidade é definido como uma propriedade sintática de tipos, o que entra em conflito com a definição padrão deste conceito, apresentada na Seção 3.3.

- A declaração de classes de tipos é obrigatória, ou seja, o tipo de símbolos sobrecarregados tem que ser explicitamente definido pelo programador. Isso entra em conflito com a idéia do mecanismo de inferência de tipos, de que tipos não precisam ser obrigatoriamente anotados.

- A declaração de classes de tipos envolve uma tarefa que não é relacionada com o reuso obtido por meio do polifmorfismo de 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.

3.9 Conclusão

Neste capítulo foram apresentadas as principais características de Haskell para o suporte a sobrecarga. Apresentamos os problemas relacionados a ambiguidade, introduzidos por classes de tipos com múltiplos parâmetros, e os principais trabalhos que propõe alternativas que procuram resolver tais problemas. Para cada uma destas alternativas, foram apresentados exemplos de utilização e foram descritas de maneira informal as restrições e os problemas decorrentes da adoção de cada uma.

No próximo capítulo, são apresentados um sistema de tipos e um algoritmo de inferência de tipos que constituem uma alternativa para suporte a adoção de classes com múltiplos parâmetros. Este sistema não adiciona nenhum mecanismo adicional para suporte a classes com múltiplos parâmetros e segue a definição padrão de ambiguidade usada na literatura.

Capítulo 4

Sistema de Tipos

4.1 Introdução

No Capítulo 3 foram apresentadas as principais características da abordagem de Haskell para sobrecarga e seus principais problemas. Neste capítulo é apresentada a definição formal de um sistema de tipos que visa solucionar, de maneira simples, o principal problema relacionado à adoção de classes de tipos com múltiplos parâmetros, a saber, ambiguidade de expressões.

Este capítulo é organizado da seguinte maneira. Primeiramente é formalizada a sintaxe da linguagem núcleo considerada (Seção 4.2). Na Seção 4.3 são apresentadas algumas propriedades de substituições. A Seção 4.4 discorre sobre contextos de tipos, a Seção 4.5 sobre provabilidade de restrições e a Seção 4.6 sobre ordens parciais sobre tipos, substituições e contextos. A Seção 4.7 apresenta detalhes sobre o algoritmo para satisfazibilidade de restrições. O algoritmo para redução de contextos (context reduction) é apresentado na Seção 4.8 e a Seção 4.9 descreve a abordagem proposta para especialização de tipos inferidos. A Seção 4.10 apresenta o sistema de tipos e a Seção 4.11 seu algoritmo de inferência. A Seção 4.12 descreve o problema de incompletude do algoritmo de inferência apresentado na Seção 4.11 em relação ao sistema de tipos descrito na Seção 4.10 e apresenta uma definição alternativa do sistema de tipos que permite-nos demonstrar corretude e completude do algoritmo em relação ao sistema de tipos. Na Seção 4.13 é apresentada uma semântica coerente, definida por indução sobre as derivações do sistema de tipos proposto na Seção 4.12. Finalmente, a Seção 4.14 descreve, de maneira sucinta, a implementação do algoritmo de inferência.

4.2 Sintaxe

4.2.1 Termos

A sintaxe dos termos da linguagem núcleo é a mesma 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:

$$\begin{array}{lll} \text{Variáveis} & x & \\ \text{Expressões} & e & ::= \quad x \mid \lambda x.\, e \mid e\, e \mid \texttt{let}\ x = e\ \texttt{in}\ e \end{array}$$

Figura 4.1. Sintaxe livre de contexto de *core-Haskell*.

4.2.2 Sintaxe de Tipos e Kinds

Tipos e *kinds* são expressos utilizando a seguinte gramática (onde o uso de metavariáveis é também indicado):

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

$$\begin{array}{llll} \text{Nome de Classe} & C \in \mathbf{C} & \text{Construtor de Tipos} & T \in \mathbf{TC} \\ \text{Variável de Tipo} & \alpha, \beta \in \mathbf{V} & \text{Conjunto de Restrições} & P, Q \end{array}$$

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 variável de expressão de tipo.

4.2.2.1 Tipos Simples e Kinds

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

$$\begin{aligned} \mathbf{V}^k &= \{\alpha \in \mathbf{V} \mid \mathrm{kind}_{\mathbf{V}}(\alpha) = k\} \\ \mathbf{TC}^k &= \{T \in \mathbf{TC} \mid \mathrm{kind}_{\mathbf{TC}}(T) = k\} \end{aligned}$$

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

$$\begin{aligned} \alpha^k &\in \mathbf{V}^k \\ T^k &\in \mathbf{TC}^k \end{aligned}$$

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

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

Consideramos que o conjunto de tipos $\mathbf{T}$ inclui somente expressões de tipo *bem formadas*, isto é, aquelas que têm um *kind* definido. Por definição, o *kind* de tipos simples é igual a $\star$.

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

A notação $[\tau]$ é usada para expressar a aplicação do construtor de listas (que possui *kind* $\star \rightarrow \star$) ao tipo τ.

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

$$tv(\mu) = \begin{cases} \{\alpha\} & \text{se } \mu = \alpha, \quad \text{para algum } \alpha \in \mathbf{V} \\ \emptyset & \text{se } \mu = T, \quad \text{para algum } T \in \mathbf{TC} \\ tv(\mu_1) \cup tv(\mu_2) & \text{se } \mu = \mu_1\,\mu_2 \end{cases}$$

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

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

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 π da forma $C\,\overline{\mu}$, é tal que cada tipo simples em $\overline{\mu}$ corresponde ao parâmetro da classe de mesma ordem na sequência (se $\overline{\mu} = \mu_1, ..., \mu_n$ então μ_1 corresponde ao primeiro parâmetro, μ_2 ao 2^o parâmetro, ..., e μ_n ao n-ésimo parâmetro da classe C).

O conjunto de variáveis de tipos de uma restrição $\pi = C\,\overline{\mu}$ é dado por:

$$tv(\pi) = tv(\overline{\mu})$$

.

Esta função pode ser estendida para conjuntos de restrições P de maneira simples:

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

4.2.2.3 Tipos

Uma expressão $\sigma = \forall\,\overline{\alpha}.\,P \Rightarrow \tau$ denota um tipo, onde $\overline{\alpha}$ denota uma sequência, possivelmente vazia, de elementos do conjunto $\{\alpha_1, ...\alpha_n\}$. Se $P = \emptyset$, dizemos que este é um tipo irrestrito, caso contrário dizemos que é um tipo com restrições. Caso $\overline{\alpha} = \emptyset$, dizemos que este é um tipo monomórfico, caso contrário dizemos que é um tipo polimórfico. As seguintes abreviações são usadas: $\forall\,\overline{\alpha}.\,\tau$, quando $P = \emptyset$; $P \Rightarrow \tau$, quando $\overline{\alpha} = \emptyset$ e τ, quando $P = \overline{\alpha} = \emptyset$.

O conjunto de variáveis livres de um tipo é definido como:

$$tv(\forall\,\overline{\alpha}.P \Rightarrow \tau) = (tv(P) \cup tv(\tau)) - \overline{\alpha}$$

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

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

4.3 Substituições

Uma substituição S é uma função de variáveis de tipo para expressões de tipo. Representamos a substituição identidade por *id*.

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

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

Neste trabalho consideramos apenas substituições S para as quais o conjunto dom(S) é finito, e tais que S preserva o *kind* das variáveis, i.e., para toda variável α, $S(\alpha^k) \in \mathtt{T}^k$.

A notação $[\alpha_1 \mapsto \mu_1, ..., \alpha_n \mapsto \mu_n]$, para $n \geq 1$ e $\alpha_i \neq \alpha_j$ para $1 \leq i < j \leq n$ denota a substituição S definida como:

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

Utilizamos a notação $[\overline{\alpha \mapsto \mu}]$ para representar $[\alpha_1 \mapsto \mu_1, ..., \alpha_n \mapsto \mu_n]$.

Substituições podem ser estendidas de maneira imediata para restrições ($S\,\pi$), conjuntos de restrições ($S\,P$), tipos restringidos ($S(P \Rightarrow \tau)$) e conjuntos de tipos restringidos, i.e. aplicando a substituição a cada variável de tipo que aparece como subtermo.

4.4 Contextos de Tipos

Um contexto de tipos contém informações sobre um programa, que podem ser utilizadas pelas regras do sistema de tipos. Neste trabalho fazemos distinção entre dois contextos, a saber:

- Contexto de tipos de variáveis: Representado pela metavariável Γ, consiste em um conjunto de associações de nomes a tipos, denotadas por $x : \sigma$, onde x é um nome e σ um tipo. Cada definição de classe

$$\begin{array}{l} \texttt{class } \forall\,\overline{\alpha}.P \Rightarrow C\,\overline{\alpha} \texttt{ where} \\ \qquad x_1 :: P_1 \Rightarrow \tau_1 \\ \qquad\qquad \vdots \\ \qquad x_n :: P_n \Rightarrow \tau_n \end{array}$$

  introduz as seguintes associações de nomes a tipos em Γ: $x_i :: \sigma_i$ onde $\sigma_i = \forall\,\overline{\alpha_i}.P \cup P_i \Rightarrow \tau_i$ e $\overline{\alpha_i} = tv(P \cup P_i \Rightarrow \tau_i)$. Definimos $\Gamma(x) = \sigma$ se $x : \sigma \in \Gamma$, e $dom(\Gamma) = \{x \mid x : \sigma \in \Gamma\}$.

- Contexto de restrições de classes e instâncias: Representado pela metavariável Θ, consiste em um par de conjuntos que contêm, respectivamente, restrições correspondentes a classes e instâncias definidas em um programa. Denotamos por Θ^{cls} o contexto de classes de um programa, que é um conjunto de restrições relativas a definições de classes e Θ^{ins} o conjunto que contém *restrições de instâncias.*

  Cada declaração de classe

$$\texttt{class } \forall\,\overline{\alpha}.P \Rightarrow C\,\overline{\alpha} \texttt{ where...}$$

introduz uma *restrição de classe* $\forall\,\overline{\alpha}.P \Rightarrow C\,\overline{\alpha}$ em Θ^{cls}. Adicionalmente, definimos $\Theta^{cls}(C) = \forall\,\overline{\alpha}.P \Rightarrow C\,\overline{\alpha}$.

Cada instância:

$$\texttt{instance}\ P \Rightarrow C\,\overline{\mu}\ \texttt{where...}$$

introduz uma *restrição de instância* $\forall\,\overline{\alpha}.P \Rightarrow C\,\overline{\mu}$ no conjunto de restrições de instâncias Θ^{ins}. Denotaremos por $\Theta^{ins}(C)$ o conjunto de restrições de instâncias definidas para uma classe C.

O contexto de tipos do escopo mais externo de um programa (escopo global) é formado pelos dois contextos de restrições, de classes e de instâncias.

4.5 Provabilidade de Restrições

Um problema central em sistemas de tipos para Haskell é determinar quando uma restrição presente em um tipo é provável. A noção de provabilidade de restrições é formalizada por uma relação entre conjuntos de restrições. Neste trabalho, consideremos uma definição de provabilidade à apresentada em [Jones, 1995a].

Dizemos que um conjunto de restrições Q é provável a partir de outro conjunto de restrições P, utilizando a informação contida no contexto Θ, se é possível provar $\Theta,\ P \Vdash Q$. A definição da relação de provabilidade é apresentada na Figura 4.3.

$$\boxed{\Theta, P \Vdash Q}$$

$$\frac{Q \subseteq P}{\Theta, P \Vdash Q}\ \texttt{Mono} \qquad \frac{\Theta, P \Vdash Q' \quad \Theta, Q' \Vdash Q}{\Theta, P \Vdash Q}\ \texttt{Trans} \qquad \frac{\Theta, P \Vdash Q}{\Theta, SP \Vdash SQ}\ \texttt{Subst}$$

$$\frac{(\forall\,\overline{\alpha}.\ P \Rightarrow C\,\overline{\mu}) \in \Theta^{ins}(C)}{\Theta, P \Vdash C\,\overline{\mu}}\ \texttt{Inst} \qquad \frac{\Theta, P \Vdash P' \quad \Theta, Q \Vdash Q'}{\Theta, P, Q \Vdash P',\ Q'}\ \texttt{Conj}$$

$$\frac{\Theta, P \Vdash C\,\overline{\alpha}' \quad \pi \in Q' \quad \forall\,\overline{\alpha}.Q' \Rightarrow C\,\overline{\alpha}' = \Theta^{cls}(C)}{\Theta, P \Vdash \pi}\ \texttt{Super}$$

Figura 4.3. Provabilidade de Restrições

As regras `Mono`, `Trans` e `Subst` representam 3 requisitos que toda relação de provabilidade de restrições deve possuir: monotonicidade, transitividade e fechamento sob substituição [Jones, 1995a]. A regra `Conj` permite combinar a prova de dois conjuntos

de restrições P e Q em uma prova de P, Q (onde $P, Q = P \cup Q$). A regra `Inst` especifica que uma determinada restrição $C\,\overline{\mu}$ é provável a partir de um conjunto de restrições P se existe uma definição de instância correspondente, isto é, $(\forall\,\overline{\alpha}.\,P \Rightarrow C\,\overline{\mu}) \in \Theta^{ins}(C)$. Finalmente, a regra `Super` permite a provabilidade de restrições utilizando-se informação sobre a hierarquia de classes presente em Θ. Uma restrição $\pi = C\,\overline{\mu}_1$ é provável a partir de um conjunto P, se existir uma classe $\forall\,\overline{\alpha}.Q' \Rightarrow C\,\overline{\alpha}'$ tal que $C\,\overline{\alpha}'$ é provável a partir de P e $\pi \in Q'$.

Exemplo 10. A fim de exemplificar o uso dessas definições, considere $\Theta = (\Theta_C, \Theta_I)$, onde $\Theta_C = \{\forall\,\alpha.\,\texttt{Eq}\ \alpha\}$ e $\Theta_I = \{\texttt{Eq Int}, \forall\,\alpha.\,\texttt{Eq}\ \alpha \Rightarrow \texttt{Eq [}\alpha\texttt{]}\}$. Temos que $\Theta \Vdash \texttt{Eq [Int]}$, como demonstrado pela seguinte derivação:

$$\dfrac{\dfrac{\texttt{Eq Int} \in \Theta^{ins}(\texttt{Eq})}{\Theta \Vdash \texttt{Eq Int}}\ \texttt{Inst} \qquad \dfrac{\dfrac{\forall\,\alpha.\,\texttt{Eq}\ \alpha \Rightarrow \texttt{Eq [}\alpha\texttt{]} \in \Theta^{ins}(\texttt{Eq})}{\Theta, \texttt{Eq}\,\alpha \Vdash \texttt{Eq [}\alpha\texttt{]}}\ \texttt{Inst} \qquad S = [\alpha \mapsto \texttt{Int}]}{\Theta, \texttt{Eq Int} \Vdash \texttt{Eq [Int]}}\ \texttt{Subst}}{\Theta \Vdash \texttt{Eq [Int]}}\ \texttt{Trans}$$

A próxima seção define ordens parciais sobre tipos. Essencialmente, as definições apresentadas são extensões das relações de [Damas & Milner, 1982] de modo a levar em consideração a provabilidade de restrições.

4.6 Ordens Parciais

Esta seção define ordens parciais e seus semi-reticulados correspondentes [Davey & Priestly, 1990] para expressões de tipos. Tais definições serão utilizadas no sistema de tipos da Figura 4.10 e para a demonstração das propriedades de uma função para computar a menor generalização comum de um conjunto de tipos [Camarão & Figueiredo, 1999].

4.6.1 Expressões de Tipos Simples

4.6.1.1 Pré-Ordem de Expressões de Tipos Simples

A relação $\preceq_{\mathbf{T}}$ é uma pré-ordem sobre o conjunto de expressões de tipos simples, de maneira que $\mu \preceq_{\mathbf{T}} \mu'$ é definido como $\mu' = S\,\mu$, para algum S.

Se $\mu \preceq_{\mathbf{T}} \mu'$, $\overline{\alpha} = tv(\mu)$ 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 $\forall\,\overline{\alpha}'.\mu'$.

4.6.1.2 Equivalência Módulo Renomeação de Variáveis

A partir da pré-ordem $\preceq_{\mathbf{T}}$ podemos definir a relação de equivalência $\equiv_{\mathbf{T}}$, de maneira que $\mu \equiv_{\mathbf{T}} \mu'$ é definido como $\mu \preceq_{\mathbf{T}} \mu'$ e $\mu' \preceq_{\mathbf{T}} \mu$

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

4.6.1.3 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 $T_{\equiv}$. Denotamos por $[\mu]_{\equiv}$ a classe de equivalência de uma expressão de tipo simples μ módulo $\equiv_{\mathbf{T}}$ ($[\mu]_{\equiv} = \{\mu' \mid \mu \equiv_{\mathbf{T}} \mu'\}$) e definimos a ordem parcial $\leq_{\mathbf{T}}$ de modo que $[\mu]_{\equiv} \leq_{\mathbf{T}} [\mu']_{\equiv}$ é definido como $\mu \preceq_{\mathbf{T}} \mu'$ (veja Lema 6 no Apêndice A).

O elemento $[\alpha]$, onde α é uma variável de tipo qualquer, possui a propriedade de ser o *elemento mínimo* da ordem parcial, i.e. para todo μ, $[\alpha] \leq_{\mathbf{T}} [\mu]$. Como qualquer elemento de uma classe de equivalência pode ser usado para representar essa classe, omitimos os colchetes e o sinal $\equiv$ quando nos referirmos a elementos de $\mathbf{T}_{\equiv}$, escrevendo μ ao invés de $[\mu]_{\equiv}$, quando ficar claro pelo contexto se estamos nos referindo a $[\mu]$ ou a μ propriamente.

4.6.2 Ordem Parcial de Tipos

Seja Σ o conjunto os tipos e Θ um contexto global contendo informações sobre restrições de classes e instâncias. Podemos definir a ordem parcial $\leq_{\Sigma}$ sobre Σ da seguinte maneira: Sejam $\sigma_1 = \forall\, \overline{\alpha}_1.\, P_1 \Rightarrow \tau_1$ e $\sigma_2 = \forall\, \overline{\alpha}_2.\, P_2 \Rightarrow \tau_2$. Definimos $\sigma_1 \leq_{\Sigma} \sigma_2$ como existe $S = [\overline{\alpha_2 \mapsto \mu}]$ tal que Θ, $P_1 \Vdash S\, P_2$ e $\tau_1 = S\, \tau_2$. Não mencionamos o contexto Θ em $\leq_{\Sigma}$ para não sobrecarregar desnecessariamente a notação.

Se $\sigma_1 \leq_{\Sigma} \sigma_2$ dizemos que σ_1 é mais geral que σ_2, ou, de maneira equivalente, que σ_2 é uma instância de σ_1. Dizemos que dois tipos σ_1 e σ_2 são equivalentes módulo renomeamento de variáveis, $\sigma_1 \equiv_{\Sigma} \sigma_2$, se $\sigma_1 \leq_{\Sigma} \sigma_2$ e $\sigma_2 \leq_{\Sigma} \sigma_1$.

4.6.3 Ordem Parcial de Substituições

Seja $\mathbb{S}$ o conjunto de todas as substituições. Definimos a ordem parcial $\leq_{\mathbb{S}}$ sobre $\mathbb{S}$ da seguinte maneira: Sejam S_1 e S_2 duas substituições quaisquer. Definimos $S_1 \leq_{\mathbb{S}} S_2$ se existe S' tal que $S_2 = S' \circ S_1$.

$$\boxed{\Theta \vdash^{\texttt{sats}} P \rightsquigarrow \mathbb{S}}$$

$$\frac{}{\Theta \vdash^{\texttt{sats}} \emptyset \rightsquigarrow \{id\}}\ \texttt{SEmpty}$$

$$\frac{\begin{array}{l}\Theta \vdash^{\texttt{sats}} \pi \rightsquigarrow \mathbb{S}_0 \\ \mathbb{S} = \{S'S \mid S \in \mathbb{S}_0,\ S' \in \mathbb{S}_1,\ \Theta \vdash^{\texttt{sats}}\ SP \rightsquigarrow \mathbb{S}_1\}\end{array}}{\Theta \vdash^{\texttt{sats}} \pi, P \rightsquigarrow \mathbb{S}}\ \texttt{SConj}$$

$$\frac{\begin{array}{l}\Delta = sat(\pi, \Theta) \\ \mathbb{S} = \{S'S \mid (S, Q, \pi') \in \Delta,\ S' \in \mathbb{S}_0,\ \Theta \vdash^{\texttt{sats}} Q \rightsquigarrow \mathbb{S}_0\}\end{array}}{\Theta \vdash^{\texttt{sats}} \{\pi\} \rightsquigarrow \mathbb{S}}\ \texttt{SInst}$$

Figura 4.4. Algoritmo para Satisfazibilidade de Restrições

4.6.4 Ordem Parcial de Contextos de Tipos

Sejam Γ e Γ' dois contextos de tipos quaisquer e Θ o contexto global contendo informações sobre símbolos sobrecarregados.Definimos uma ordem parcial sobre contextos de tipos, $\leq_\omega$, da seguinte maneira: $\Gamma \leq_\omega \Gamma'$ se $\Gamma(x) \leq_\Sigma \Gamma'(x)$, para todo $x \in dom(\Gamma)$.

4.7 Satisfazibilidade de Restrições

O conjunto de restrições P em um tipo $\sigma = (\forall\, \overline{\alpha}.P \Rightarrow \tau)$ restringe o conjunto de tipos para os quais σ pode ser instanciado em um dado contexto.

O conceito de satisfazibilidade de restrições pode ser formalizado utilizando-se a relação de provabilidade. Dizemos que um conjunto de restrições P é satisfazível em um contexto Θ se existe uma substituição S tal que $S\,P$ é provável em Θ, isto é, se $\Theta \Vdash S\,P$.

A Figura 4.4 apresenta um algoritmo que, caso termine, retorna o conjunto de substituições que satisfazem um dado conjunto de restrições P. Uma vez que o problema de satisfazibilidade de um conjunto de restrições em um contexto é indecidível [Volpano & Smith, 1991], o algoritmo da Figura 4.4 pode não terminar.

O algoritmo é definido como um conjunto de regras para da forma $\Theta \vdash^{\texttt{sats}} P \rightsquigarrow \mathbb{S}$, onde $\mathbb{S}$ é o conjunto de substituições que satisfazem P em Θ. A seguinte função é utilizada na definição do algoritmo de satisfazibilidade:

$$
\begin{aligned}
sat(\pi, \Theta) = \ \{(S|_{tv(\pi)}, SP, \pi_0) \mid \ & (\forall\, \overline{\alpha}.\, P_0 \Rightarrow \pi_0) \in \Theta, \\
& S_1 = [\overline{\alpha} \mapsto \overline{\beta}],\ \overline{\beta} \text{ são novas variáveis} \\
& (P \Rightarrow \pi') = S_1\,(P_0 \Rightarrow \pi_0) \\
& S = \ mgu(\pi = \pi')\}
\end{aligned}
$$

A função $sat(\pi, \Theta)$ retorna informações sobre as instâncias presentes em Θ que unificam com a restrição π. Para cada uma dessas instâncias, a função *sat* retorna uma tripla constituída da substituição obtida pela unificação de π com esta instância, as restrições e a cabeça desta instância.

A interpretação das regras da Figura 4.4 é direta. A regra `SEmpty` especifica que um conjunto vazio de restrições é satisfeito pelo conjunto contendo uma apenas a substituição identidade, denotada por *id*. A regra `SInst` calcula o conjunto $\mathbb{S}_0$ de substituições que satisfazem π, ou seja o conjunto de instâncias $\forall\, \overline{\alpha}.\, P_0 \Rightarrow \pi_0 \in \Theta$ tais que π unifica com π_0, e compõe essas substituições com aquelas que satisfazem as restrições introduzidas por estas instâncias. Finalmente, a regra `SConj` lida com conjuntos com mais de uma de restrição.

Os seguintes exemplos podem ajudar a esclarecer o comportamento do algoritmo.

Exemplo 11. Considere $P = \{A\, a\, b,\ D\, b\}$ e

$$\Theta = \{A\ Int\ [Int]\,, A\ Int\ [Bool]\,, C\ Int, \forall b.\, C\, b \Rightarrow D\ [b]\,\}$$

A satisfazibilidade de P em relação Θ produz o seguinte conjunto de substituições $\mathbb{S}$:

$$\frac{\begin{array}{l}\Theta \vdash^{\texttt{sats}} A\, a\, b \rightsquigarrow \mathbb{S}_0 \\ \mathbb{S} = \{S'S \mid S \in \mathbb{S}_0,\ S' \in \mathbb{S}_1, \Theta \vdash^{\texttt{sats}} \{S(D\, b)\} \rightsquigarrow \mathbb{S}_1\,\}\end{array}}{\Theta \vdash^{\texttt{sats}} A\, a\, b, \{D\, b\} \rightsquigarrow \mathbb{S}}\ \texttt{SConj}$$

então:

$$\frac{\begin{array}{l}\Delta_0 = \{(S_1, \emptyset, A\ Int\ [Int]\,), (S_2, \emptyset, A\ Int\ [Bool]\,)\} \\ \mathbb{S}_0 = \{S'S \mid (S, Q, \pi') \in \Delta_0,\ S' \in \mathbb{S}', \Theta \vdash^{\texttt{sats}} Q \rightsquigarrow \mathbb{S}'\,\}\end{array}}{\Theta \vdash^{\texttt{sats}} A\, a\, b \rightsquigarrow \mathbb{S}_0}\ \texttt{SInst}$$

onde $S_1 = [a \mapsto Int,\, b \mapsto [Int]\,]$, $S_2 = [a \mapsto Int,\, b \mapsto [Bool]\,]$.

Pela regra `SConj`, o conjunto de substituições para $S_1(D\ b) = D\ [Int]$ and

$S_2(D\ b) = D\ [Bool]$ é dado por:

$$\frac{\begin{array}{l}\Delta_1 = \{(S_1'|_\emptyset, \{C\ Int\}, D\ [b])\} \\ \mathbb{S}_1^1 = \{S'S \mid (S, Q, \pi') \in \Delta_1,\ S' \in \mathbb{S}', \Theta \vdash^{\mathtt{sats}} Q \rightsquigarrow \mathbb{S}'\}\end{array}}{\Theta \vdash^{\mathtt{sats}} D\ [Int] \rightsquigarrow \mathbb{S}_1^1}\ \mathtt{SInst}$$

onde $S_1' = [b_1 \mapsto Int]$, b_1 é uma nova variável, $S_1'|_\emptyset = id$, e

$$\frac{\begin{array}{l}\Delta_2 = \{(S_2'|_\emptyset, \{C\ Bool\}, D\ [b])\} \\ \mathbb{S}_1^2 = \{S'S \mid (S, Q, \pi') \in \Delta_2, S' \in \mathbb{S}', \Theta \vdash^{\mathtt{sats}} Q \rightsquigarrow \mathbb{S}'\}\end{array}}{\Theta \vdash^{\mathtt{sats}} D\ [Bool] \rightsquigarrow \mathbb{S}_1^2}\ \mathtt{SInst}$$

onde $S_2' = [b_2 \mapsto Bool]$, b_2 é uma nova variável, $S_2'|_\emptyset = id$. Now, $\mathbb{S}_1^1 = \{id\}$ and $\mathbb{S}_1^2 = \emptyset$. Assim, temos que $\mathbb{S} = \{S_1\}$.

O próximo exemplo, retirado de [Camarão et al., 2004], ilustra a não terminação do cálculo da satisfazibilidade de um conjunto de restrições. Neste exemplo, a notação $T^2\,\tau$ abrevia $T(T\,\tau)$ e similarmente para índices maiores que 2.

Exemplo 12. Seja $\Theta = \{\forall a, b.\, \{C\ a\ b\} \Rightarrow C\ (T^2\ a)\ b\}$ e considere a computação da satisfazibilidade de $\pi = C\ a\ (T\ a)$ em relação a Θ. Temos que π unifica com a cabeça da instância $\forall a, b.\, (C\ a\ b) \Rightarrow C\ (T^2\ a)\ b$, produzindo a substituição $S = [a \mapsto T^2\,a_1,\ b_1 \mapsto T^3\,a_1]$. Então deve-se computar recursivamente a satisfazibilidade de $S(C\ a_1\ b_1) = C\ a_1\ (T^3\ a_1)$. Mas este também unifica com $\forall a, b.\, (C\ a\ b) \Rightarrow C\ (T^2\ a)\ b$, produzindo $S_1 = [a_1 \mapsto (T^2\ a_2), b_2 \mapsto (T^3 a_1 = T^5\ a_2)]$. Então, novamente, devemos computar a satisfazibilidade de $S_1(C\ a_2\ b_2) = C\ a_2\ (T^5\ a_2)$, repetindo todo o processo e ocasionando a não terminação.

Propriedades do algoritmo de satisfazibilidade são enunciadas e demonstradas no Apêndice A. A próxima seção descreve, de maneira suscinta, abordagens para garantir a terminação de testes para satisfazibilidade descritas na literatura e apresenta uma nova versão do algoritmo descrito nesta seção, em que são impostas restrições para garantir sua terminação.

4.7.1 Critérios de Terminação

Para garantir a terminação do teste de satisfazibilidade, os artigos [Stuckey & Sulzmann, 2005, Sulzmann et al., 2006a] usam diversas restrições so-

bre declarações de classes e instância. Estas restrições, conhecidas como "Paterson Conditions", são as seguintes:

- O contexto de uma declaração de classe deve ser formado apenas por variáveis de tipos e não construtores de tipos, e cada restrição presente neste contexto deve ser formada por variáveis de tipos distintas.

- Para cada instância `instance` $P \Rightarrow \pi$:

  - Nenhuma variável de tipo deve ocorrer mais vezes em uma restrição presente em P do que na cabeça π.

  - A soma do número de ocorrências de construtores e de variáveis de tipos em P deve ser menor do que em π.

- Declarações de instâncias não devem ser sobrepostas. Isto é, para qualquer par de instâncias `instance` $P_1 \Rightarrow \pi_1$ e `instance` $P_2 \Rightarrow \pi_2$ não deve existir uma substituição S tal que $S\,\pi_1 = S\,\pi_2$.

O compilador Haskell GHC [S. P. Jones and others, 1998] utiliza essas restrições para garantir a terminação de testes de satisfazibilidade e de redução de contexto (Seção 4.8). Porém, como essas restrições não permitem a definição de diversas declarações de instâncias de interesse prático, o compilador GHC provê diretivas que permitem desabilitar tais restrições. Neste caso, a terminação é garantida impondo-se um limite no número de chamadas recursivas realizadas na execução do algoritmo de satisfazibilidade.

Em [Camarão et al., 2004] é apresentado um algoritmo de inferência de tipos incompleto para o problema, baseado no Sistema CT [Camarão et al., 2007, Camarão & Figueiredo, 1999]. Tal proposta também utiliza um limite para o número de chamadas recursivas realizadas pelo algoritmo para garantir a terminação.

Neste trabalho, propomos uma condição de terminação que, em vez de usar um limite para o número de chamadas recursivas ou restringir a forma de definições de classes e instâncias, considera que uma restrição π é satisfazível em um contexto Θ se, e somente se existir uma cadeia descrescente finita de restrições tal que, para todo par de restrições π_i, π_j consecutivas nesta cadeia, ou existe uma "medida" associada a π_j que é menor que a medida de π_i, ou existe k tal que $\pi_i = C\,\mu_1 ... \mu_k ... \mu_n$ e $\pi_j = C\,\mu'_1 ... \mu'_k ... \mu'_n$ e a medida de μ'_k é menor do que μ_k. Esta idéia é formalizada a seguir.

Seja $\eta(\pi)$ a medida de complexidade de uma restrição definida como o número de variáveis e construtores de tipos. Mais formalmente:

$$\begin{array}{ll} \eta(C\,\mu_1 \cdots \mu_n) & = \sum_{i=1}^{n} \eta(\mu_i) \\ \eta(T) & = 1 \\ \eta(\alpha) & = 1 \\ \eta(\mu_1\,\mu_2) & = \eta(\mu_1) + \eta(\mu_2) \end{array}$$

Para garantir que a satisfazibilidade de uma restrição siga uma cadeia decrescente de medidas de complexidade, o algoritmo utiliza um mapeamento finito Φ entre as cabeças de instâncias em Θ e pares (I, Π).

O primeiro componente deste par, I, é uma tupla de valores inteiros $(v_0, ..., v_n)$, onde v_0 é o menor $\eta(S\,\pi')$ de todas as restrições π' que unificaram com uma cabeça de instância π_0 durante o teste de satisfazibilidade de π, onde $S = mgu(\pi', \pi_0)$. Cada v_i, $1 \leq i \leq n$, é o menor $\eta(\mu_i)$ onde μ_i é um tipo pertencendo a algum $S\,\pi'$ que unificou com π_0. Utilizaremos a notação $I.v_i$ para representar o i-ésimo valor de I e, similarmente, $\Phi(\pi_0).I$ e $\Phi(\pi_0).\Pi$ para denotar o primeiro e segundo componente de $\Phi(\pi_0)$, respectivamente.

O segundo componente, Π, de $\Phi(\pi_0)$ contém restrições π' que unificaram com π_0 e possuem medidas menores ou iguais a $\Phi(\pi_0).I.v_0$. Isto é necessário para permitir que restrições distintas com medidas iguais unifiquem com π_0. O Exemplo 16 ilustra essa situação. Para o cálculo da satisfazibilidade é usada a notação $\Phi[\pi_0, \pi]$ para a atualização do valor de $\Phi(\pi_0)$, definida como:

$$\Phi[\pi_0, \pi] = \begin{cases} \mathit{Fail} & \text{se } v'_i = -1 \text{ para } i = 0, \ldots, n \\ \Phi' & \text{caso contrário} \end{cases}$$

$$\begin{array}{rll} \text{onde} & \Phi'(\pi_0) & = ((v'_0, v'_1, \ldots, v'_n), \Phi(\pi_0).\Pi \cup \{\pi\}) \\ & \Phi'(x) & = \Phi(x) \text{ para } x \neq \pi_0 \end{array}$$

$$v'_0 = \begin{cases} \eta(\pi) & \text{se } \eta(\pi) < \Phi(\pi_0).v_0 \text{ ou} \\ & \eta(\pi) = \Phi(\pi_0).v_0 \text{ e } \pi \notin \Phi(\pi_0).\Pi \\ -1 & \text{caso contrário} \end{cases}$$

$$\text{para } i = 1, \ldots, n \qquad v'_i = \begin{cases} \eta(\mu_i) & \text{se } \eta(\mu_i) < \Phi(\pi_0).v_i \\ -1 & \text{caso contrário} \end{cases}$$

A função `tsat` para cálculo de satisfazibilidade é definida na Figura 4.5 como um conjunto de regras para derivar $\Theta, \Phi \vdash^{\texttt{tsat}} P \rightsquigarrow \mathbb{S}$, utilizando Φ como parâmetro

adicional. O conjunto de substituições que satisfazem ao conjunto de restrições P, com respeito a Θ, é dado por $\mathbb{S}$, tal que $\Theta, \Phi_0 \vdash^{\texttt{tsat}} P \rightsquigarrow \mathbb{S}$ é provável, onde $\Phi_0(\pi_0) = (I_0, \emptyset)$ para cada cabeça de instância π_0 em Θ e, sendo π_0 uma instância de n parâmetros, I_0 é uma tupla de $(n+1)$ valores, todos iguais a uma constante inteira suficientemente grande que representamos por ∞.

$$\boxed{\Theta, \Phi \vdash^{\texttt{tsat}} P \rightsquigarrow \mathbb{S}}$$

$$\frac{}{\Theta, \mathit{Fail} \vdash^{\texttt{tsat}} P \rightsquigarrow \emptyset}\ \texttt{SFail}_1 \qquad \frac{}{\Theta, \Phi \vdash^{\texttt{tsat}} \emptyset \rightsquigarrow \{id\}}\ \texttt{SEmpty}_1$$

$$\frac{\begin{array}{l}\Theta, \Phi \vdash^{\texttt{tsat}} \pi \rightsquigarrow \mathbb{S}_0 \\ \mathbb{S} = \{S'S \mid S \in \mathbb{S}_0,\ S' \in \mathbb{S}_1,\ \Theta, \Phi \vdash^{\texttt{tsat}} SP \rightsquigarrow \mathbb{S}_1\}\end{array}}{\Theta, \Phi \vdash^{\texttt{tsat}} \pi, P \rightsquigarrow \mathbb{S}}\ \texttt{SConj}_1$$

$$\frac{\begin{array}{l}\Delta = \mathit{sat}(\pi, \Theta) \\ \mathbb{S} = \{S'S \mid (S, Q, \pi') \in \Delta,\ S' \in \mathbb{S}_0,\ \Theta, \Phi[\pi', \pi] \vdash^{\texttt{tsat}} Q \rightsquigarrow \mathbb{S}_0\}\end{array}}{\Theta, \Phi \vdash^{\texttt{tsat}} \pi \rightsquigarrow \mathbb{S}}\ \texttt{SInst}_1$$

Figura 4.5. Algoritmo para Satisfazibilidade com Critério de Terminação

Nos próximos exemplos, `B`, `I` and `F` são utilizados como abreviações para *Bool*, *Int* and *Float*, respectivamente.

Exemplo 13. Considere o cálculo da satisfazibilidade de $\pi = \texttt{Eq[[I]]}$ em $\Theta = \{Eq\ \texttt{I},\ \forall a.\ Eq\ a \Rightarrow\ \texttt{Eq[}a\texttt{]}\}$, sendo $\pi_0 =\ \texttt{Eq[}a\texttt{]}$; temos que:

$$\frac{\begin{array}{l}\Delta_0 = \mathit{sat}(\pi, \Theta) = \{\big(S|_\emptyset, \{\ \texttt{Eq[I]}\}, \pi_0\big)\} \\ S = [a_1 \mapsto \texttt{[I]}] \\ \mathbb{S}_0 = \{S_1 \circ id \mid\ S_1 \in \mathbb{S}_1,\ \ \Theta, \Phi_1 \vdash^{\texttt{tsat}} \texttt{Eq[I]} \rightsquigarrow \mathbb{S}_1\}\end{array}}{\Theta, \Phi_0 \vdash^{\texttt{tsat}} \pi \rightsquigarrow \mathbb{S}_0}$$

onde $\Phi_1 = \Phi_0[\pi_0, \pi]$, onde $\Phi_1 = ((3,3), \{\pi\})$ e a_1 é uma nova variável de tipos; então:

$$\frac{\begin{array}{l}\Delta_1 = \mathit{sat}(Eq\,\texttt{[I]}, \Theta) = \{\big(S'|_\emptyset, \{\ Eq\ \texttt{I}\}, \pi_0\big)\} \\ S' = [a_2 \mapsto \texttt{I}] \\ \mathbb{S}_1 = \{S_2 \circ id \mid\ S_2 \in \mathbb{S}_2,\ \ \Theta, \Phi_2 \vdash^{\texttt{tsat}}\ \ Eq\,\texttt{I} \rightsquigarrow \mathbb{S}_2\}\end{array}}{\Theta, \Phi_1 \vdash^{\texttt{tsat}}\ \ Eq\,\texttt{[I]} \rightsquigarrow \mathbb{S}_1}$$

onde $\Phi_2 = \Phi_1[\pi_0, Eq\,\texttt{[I]}]$ (com $\Phi_2(\pi_0) = ((2,2), \Pi_2)$, onde $\Pi_2 = \{\pi,\ Eq\,\texttt{[I]}\}$), já que $\eta(Eq\,\texttt{[I]}) = 2$ é menor que $\Phi_1(\pi_0).I.v_0 = 3$); então:

$$\frac{\begin{array}{l}\Delta_2 = sat(\ Eq\,\texttt{I}, \Theta) = \{\big(id, \emptyset,\ Eq\,\texttt{I}\big)\} \\ \mathbb{S}_2 = \{S_3 \circ id \mid\ S_3 \in \mathbb{S}_3,\ \ \Theta, \Phi_3 \vdash^{\texttt{tsat}} \emptyset \rightsquigarrow \mathbb{S}_3 = \{id\}\}\end{array}}{\Theta, \Phi_2 \vdash^{\texttt{tsat}}\ \ Eq\,\texttt{I} \rightsquigarrow \mathbb{S}_2}$$

onde $\Phi_3 = \Phi_2[Eq\,\texttt{I},\ Eq\,\texttt{I}]$, $\mathbb{S}_3 = \{id\}$ por ($\texttt{SEmpty}_1$).

Exemplo 14. Considere novamente o Examplo 12, no qual desejamos obter o conjunto de substituições que satisfazem à restrição $\pi = C\ a\ (T\ a)$, dado $\Theta = \{\forall a, b.\, C\ a\ b \Rightarrow C\ (T^2\ a)\ b\}$. O cálculo da satisfazibilidade de π pela função definida na Figura 4.4 não termina. Seja $\pi_0 = C\ (T^2\ a)\ b$. Temos:

$$\frac{\begin{array}{l}\Delta_0 = sat(\pi, \Theta) = \{\big(S\,|_{\{a\}}, \{\pi_1\}, \pi_0\big)\} \\ S = [a \mapsto T^2\ a_1, b_1 \mapsto T^3\ a_1] \\ \pi_1 = C\ a_1\ (T^3\ a_1) \\ \mathbb{S}_0 = \{S_1 \circ [a \mapsto T^2\ a_1] \mid S_1 \in \mathbb{S}_1,\ \ \Theta, \Phi_1 \vdash^{\texttt{tsat}} \pi_1 \rightsquigarrow \mathbb{S}_1\}\end{array}}{\Theta, \Phi_0 \vdash^{\texttt{tsat}} \pi \rightsquigarrow \mathbb{S}_0}$$

onde $\Phi_1 = \Phi_0[\pi_0, S\pi]$, $\eta(S\pi) = \eta(C\ (T^2\ a_1)\ (T^3\ a_1)) = 7 < \Phi_0(\pi_0).I.v_0 = \infty$; então:

$$\frac{\begin{array}{l}\Delta_1 = sat(\pi_1, \Theta) = \{\big(S'\,|_{\{a_1\}},\ \{\pi_2\}, \pi_0\big)\} \\ S' = [a_1 \mapsto T^2\ a_2, b_2 \mapsto T^3\ a_1 = T^5 a_2] \\ \pi_2 = C\ a_2\ (T^5\ a_2) \\ \mathbb{S}_1 = \{S_2 \circ [a_1 \mapsto T^2\ a_2] \mid S_2 \in \mathbb{S}_2,\ \ \Theta, \Phi_2 \vdash^{\texttt{tsat}} \pi_2 \rightsquigarrow \mathbb{S}_2\}\end{array}}{\Theta, \Phi_1 \vdash^{\texttt{tsat}} \pi_1 \rightsquigarrow \mathbb{S}_1}$$

onde $\Phi_2 = \Phi_1[\pi_0, S'\pi_1]$, $S'\pi_1 = C\ (T^2\ a_2)\ (T^5\ a_2)$ e, uma vez que $\eta(S'\pi_1) = 9 > \Phi_1(\pi_0).I.v_0 = 7$, temos que $\Phi_2(\pi_0).I = (-1, \eta(T^2\ a_2) = 3, \eta(T^5\ a_2) = 6)$; então:

$$\frac{\begin{array}{l}\Delta_1 = sat(\pi_2, \Theta) = \{\big(S'\,|_{\{a_2\}},\ \{\pi_3\}, \pi_0\big)\} \\ S'' = [a_2 \mapsto T^2\ a_3, b_3 \mapsto T^5\ a_2 = T^7 a_3] \\ \pi_3 = C\ a_3\ (T^7\ a_3) \\ \mathbb{S}_2 = \{S_3 \circ [a_2 \mapsto T^2\ a_3] \mid S_3 \in \mathbb{S}_3,\ \ \Theta, \Phi_3 \vdash^{\texttt{tsat}} \pi_3 \rightsquigarrow \mathbb{S}_3\}\end{array}}{\Theta, \Phi_1 \vdash^{\texttt{tsat}} \pi_1 \rightsquigarrow \mathbb{S}_2}$$

onde $\Phi_3 = \Phi_2[\pi_0, S''\pi_2] = Fail$, já que $\eta(S''\pi_2) = \eta(C\ (T^3\ a_3)\ (T^7\ a_3)) = 12 > \Phi_2(\pi_0).I.v_0 = 9$ e não existe i tal que $\Phi_3(\pi_0).I.v_i \neq -1$, o que significa que não há parâmetro de $S''\pi_2$ com um valor η sempre decrescente.

O próximo exemplo apresenta um conjunto satisfazível no qual a computação da satisfazibilidade envolve a computação da satisfazibilidade de restrições π' que unificam com uma cabeça de instância π_0 tal que $\eta(\pi')$ é maior que o valor v_0 superior associado a π_0.

Exemplo 15. Considere o cálculo da satisfazibilidade de $\pi = C\,\mathtt{I}\,(T^3\,\mathtt{I})$ em $\Theta = \{C\,(T\,a)\,\mathtt{I}, \forall a, b.\, C\,(T^2\,a)\,b \Rightarrow C\,a\,(T\,b)\}$. Onde $\pi_0 = C\,a\,(T\,b)$, temos:

$$\frac{\begin{array}{l}\Delta_0 = sat(\pi, \Theta) = \{(S\,|_\emptyset, \{\pi_1\}, \pi_0)\} \\ S = [a_1 \mapsto \mathtt{I}, b_1 \mapsto T^2\,\mathtt{I}] \\ \pi_1 = C\,(T^2\,\mathtt{I})\,(T^2\,\mathtt{I}) \\ \mathbb{S}_0 = \{S_1 \circ id \mid S_1 \in \mathbb{S}_1, \quad \Theta, \Phi_1 \vdash^{\mathtt{tsat}} \pi_1 \rightsquigarrow \mathbb{S}_1\}\end{array}}{\Theta, \Phi_0 \vdash^{\mathtt{tsat}} \pi \rightsquigarrow \mathbb{S}_0}$$

onde $S\pi = \pi$ e $\Phi_1 = \Phi_0[\pi_0, \pi]$ (produzindo $\Phi_1(\pi_0).I = (5, 1, 4)$); então:

$$\frac{\begin{array}{l}\Delta_1 = sat(\pi_1, \Theta) = \{(S'\,|_\emptyset, \{\pi_2\}, \pi_0)\} \\ S' = [a_2 \mapsto T^2\,\mathtt{I}, b_2 \mapsto T\,\mathtt{I}] \\ \pi_2 = C\,(T^4\,\mathtt{I})\,(T\,\mathtt{I}) \\ \mathbb{S}_1 = \{S_2 \circ [a_1 \mapsto T^2\,a_2] \mid S_2 \in \mathbb{S}_2, \;\; \Theta, \Phi_2 \vdash^{\mathtt{tsat}} \pi_2 \rightsquigarrow \mathbb{S}_2\}\end{array}}{\Theta, \Phi_1 \vdash^{\mathtt{tsat}} \pi_1 \rightsquigarrow \mathbb{S}_1}$$

onde $S'\pi_1 = \pi_1$ e $\Phi_2 = \Phi_1[\pi_0, \pi_1]$. Já que $\eta(\pi_1) = 6 > 5 = \Phi_1(\pi_0).I.v_0$, então $\Phi_2(\pi_0).I = (-1, -1, 3)$.

Novamente, π_2 unifica com π_0, produzindo $S' = [a_3 \mapsto T^4\,\mathtt{I}, b_2 \mapsto \mathtt{I}]$ (assim $S'\pi_2 = \pi_2$, $\Phi_3 = \Phi_2[\pi_0, \pi_2]$ com $\Phi_3(\pi_0).I = (-1, -1, 2)$). Finalmente, a satisfazibilidade é testada para $\pi_3 = C\,(T^6\,\mathtt{I})\mathtt{I}$, que unifica com $C\,(T\,a)\,\mathtt{I}$, retornando $\mathbb{S}_3 = \{[a_3 \mapsto T^5\,\mathtt{I}]|_\emptyset\} = \{id\}$. Portanto, a restrição π é satisfazível, com $\mathbb{S}_0 = \{id\}$.

O próximo exemplo ilustra o uso do conjunto de restrições Π como parte da função Φ.

Exemplo 16. Considere a satisfazibilidade de $\pi = C\,(T^2\,\mathtt{I})\,\mathtt{F}$, onde $\pi_0 = C\,(T\,a)\,b$ e $\Theta = \{C\,\mathtt{I}\,(T^2\,\mathtt{F}), \forall a, b.\, C\,a\,(T\,b) \Rightarrow C\,(T\,a)\,b\}$:

$$\frac{\begin{array}{l}\Delta_0 = sat(\pi, \Theta) = \{(S\,|_\emptyset, \{\pi_1\}, \pi_0)\} \\ S = [a_1 \mapsto (T\,\mathtt{I}), b_1 \mapsto \mathtt{F}], \;\; \pi_1 = C\,(T\,\mathtt{I})\,(T\,\mathtt{F}) \\ \mathbb{S}_0 = \{S_1 \circ id \mid \;\; S_1 \in \mathbb{S}_1, \quad \Theta, \Phi_1 \vdash^{\mathtt{tsat}} \pi_1 \rightsquigarrow \mathbb{S}_1\}\end{array}}{\Theta, \Phi_0 \vdash^{\mathtt{tsat}} \pi \rightsquigarrow \mathbb{S}_0}$$

onde $S\pi = \pi$ e $\Phi_1 = \Phi_0[\pi_0, \pi] = ((4, 3, 1), \{\pi\})$; então:

$$\frac{\begin{array}{l}\Delta_1 = sat(\pi_1, \Theta) = \{\big(S' \mid_\emptyset, \{\pi_2\}, \pi_0\big)\} \\ S' = [a_2 \mapsto \ \mathtt{I}, b_2 \mapsto T\,\mathtt{F}], \ \ \pi_2 = C\,\mathtt{I}\,(T^2\,\mathtt{F}) \\ \mathbb{S}_1 = \{S_2 \circ id \mid \ S_2 \in \mathbb{S}_2, \ \ \Theta, \Phi_2 \vdash^{\mathtt{tsat}} \pi_2 \rightsquigarrow \mathbb{S}_2\}\end{array}}{\Theta, \Phi_1 \vdash^{\mathtt{tsat}} \pi_1 \rightsquigarrow \mathbb{S}_1}$$

onde $S'\pi_1 = \pi_1$ and $\Phi_2 = \Phi_1[\pi_0, \pi_1]$. Já que $\eta(\pi_1) = 4 = \Phi_1(\pi_0).I.v_0$ e π_1 não pertence a $\Phi_1(\pi_0).\Pi$, obtemos que $\mathbb{S}_2 = \{id\}$ e π é satisfazível (uma vez que $sat(C\,\mathtt{I}\,(T^2\,\mathtt{F}), \Theta) = \{(id, \emptyset, C\,\mathtt{I}\,(T^2\,\mathtt{F})\})$.

Uma vez que o problema de satisfazibilidade de restrições é, em geral, indecidível [Smith, 1991], existem instâncias deste problema para as quais o algoritmo proposto incorretamente reporta insatisfazibilidade. Em todos os exemplos mencionados na literatura [Sulzmann et al., 2006a, Stuckey & Sulzmann, 2005] e casos de teste do compilador GHC (envolvendo as extensões pertinentes) o algoritmo comporta-se como esperado.

Propriedades relevantes do algoritmo de satisfazibilidade apresentado nesta seção são definidas e provadas no Apêndice A.

4.8 Redução de Contexto

O processo de simplificação de um conjunto de restrições, conhecido como *redução de contexto*, consiste em transformar cada restrição π deste conjunto para o conjunto de restrições obtido, recursivamente, pela redução do contexto P introduzido pela instância que "casa" com π, até que $P = \emptyset$ ou não exista tal instância. Neste último caso, considera-se que π reduz para si próprio. Dizemos que uma instância $\forall\,\overline{\alpha}.\,P \Rightarrow \pi'$ casa com uma restrição π se existe uma substituição S tal que $S\,\pi' = \pi$.

Nesta seção é definido um algoritmo para redução de um conjunto de restrições que utiliza o critério de terminação proposto na seção 4.7.1. Como um exemplo simples de não terminação, considere a redução da restrição $C\,a$ quando $\Theta = \{\forall a.\,C\,a \Rightarrow C\,a\}$.

Antes de apresentar o algoritmo proposto, faz-se necessária a definição de uma função para coletar informações sobre instâncias que casam com uma determinada restrição em um contexto.

$$
\begin{array}{rl}
\mathit{matches}(\pi, \Theta) & = \{(S\,P, \pi') \mid (S, S\,P, \pi') \in \mathit{sats}\big([\overline{\alpha} \mapsto \overline{K}]\pi, \Theta\big)\} \\
& \textit{onde:} \\
& \quad \overline{\alpha} = \mathit{tv}(\pi) \text{ and} \\
& \quad \overline{K} \text{ são novas constantes de Skolem}
\end{array}
$$

Como neste trabalho consideramos que Θ não possui instâncias sobrepostas, toda chamada de $\mathit{matches}(\pi, \Theta)$ é sempre um conjunto unitário ou vazio.

O processo de redução de um conjunto P pode ser descrito pela seguinte definição, que utiliza a função apresentada na figura 4.6.

$$
\frac{\text{for } i = 1, \ldots, n,\ Q_i = \begin{cases} \pi_i & \text{if } \Theta, \Phi_0 \vdash^{\mathtt{simp}} \pi_i \rightsquigarrow \mathit{fail} \\ Q'_i & \text{if } \Theta, \Phi_0 \vdash^{\mathtt{simp}} \pi_i \rightsquigarrow Q'_i \end{cases}}{\Theta \vdash^{\mathtt{simp}_0} \{\pi_1, \ldots \pi_n\} \rightsquigarrow Q_1, \ldots, Q_n}\ \mathtt{R}_0
$$

Intuitivamente, reduzir um conjunto de restrições consiste em reduzir cada uma delas separadamente realizando a união dos conjuntos resultantes da redução.

$$\boxed{\Theta \vdash^{\mathtt{simp}} P \rightsquigarrow Q}$$

$$
\frac{}{\Theta, \Phi \vdash^{\mathtt{simp}} \emptyset \rightsquigarrow \emptyset}\ \mathtt{REmpty}
\qquad
\frac{\mathit{matches}(\pi, \Theta) = \emptyset}{\Theta, \Phi \vdash^{\mathtt{simp}} \pi \rightsquigarrow \pi}\ \mathtt{RStop}
$$

$$
\frac{}{\Theta, \mathit{Fail} \vdash^{\mathtt{simp}} \pi \rightsquigarrow \mathit{fail}}\ \mathtt{RFail}
\qquad
\frac{\Theta, \Phi \vdash^{\mathtt{simp}} \pi \rightsquigarrow P \quad \Theta, \Phi \vdash^{\mathtt{simp}} Q \rightsquigarrow Q'}{\Theta, \Phi \vdash^{\mathtt{simp}} \pi, Q \rightsquigarrow P, Q'}\ \mathtt{RConj}_1
$$

$$
\frac{\Theta, \Phi \vdash^{\mathtt{simp}} \pi \rightsquigarrow \mathit{fail}}{\Theta, \Phi \vdash^{\mathtt{simp}} \pi, Q \rightsquigarrow \mathit{fail}}\ \mathtt{RConj}_2
\qquad
\frac{\Theta, \Phi \vdash^{\mathtt{simp}} \pi \rightsquigarrow P \quad \Theta, \Phi \vdash^{\mathtt{simp}} Q \rightsquigarrow \mathit{fail}}{\Theta, \Phi \vdash^{\mathtt{simp}} \pi, Q \rightsquigarrow \mathit{fail}}\ \mathtt{RConj}_3
$$

$$
\frac{\{(P, \pi')\} = \mathit{matches}(\pi, \Theta) \quad \Theta, \Phi[\pi', \pi] \vdash^{\mathtt{simp}} P \rightsquigarrow Q}{\Theta \vdash^{\mathtt{simp}} \pi \rightsquigarrow Q}\ \mathtt{RInst}_1
\qquad
\frac{\{(P, \pi')\} = \mathit{matches}(\pi, \Theta) \quad \Theta, \Phi[\pi', \pi] \vdash^{\mathtt{simp}} P \rightsquigarrow \mathit{fail}}{\Theta \vdash^{\mathtt{simp}} \pi \rightsquigarrow \mathit{fail}}\ \mathtt{RInst}_2
$$

Figura 4.6. Algoritmo para Redução de Contexto

As regras da figura 4.6 são análogas às apresentadas na figura 4.5, exceto que o não atendimento do critério de terminação é propagado pelas chamadas recursivas do

algoritmo, para indicar que a restrição atual não pode ser reduzida. Regras adicionais fazem-se necessárias para tratar a propagação de falhas no processo de redução, sendo a interpretação de cada uma delas bastante simples.

O seguinte exemplo ilustra o comportamento do algoritmo de redução.

Exemplo 17. Considere $\Theta = \{\forall a.\, C\,(T\,a) \Rightarrow C\,a,\ D\,\mathtt{I}\}$ e $P = \{D\,\mathtt{I},\ C\,a\}$. De acordo com a regra (R_0), reduzir P consiste em reduzir as restrições $D\,\mathtt{I}$ e $C\,a$ separadamente.

A redução de $D\,\mathtt{I}$ é definida pela regra $(\mathtt{RInst}_1)$:

$$\frac{\{(\emptyset, D\,\mathtt{I})\} = \mathit{matches}(D\,\mathtt{I}, \Theta) \quad \Theta, \Phi_0[D\,\mathtt{I}, D\mathtt{I}] \vdash^{\mathtt{simp}} \emptyset \rightsquigarrow \emptyset}{\Theta, \Phi_0 \vdash^{\mathtt{simp}} D\,\mathtt{I} \rightsquigarrow \emptyset}$$

Por sua vez, a redução de $\pi = \pi_0 = C\,a$ resulta em falha:

$$\frac{\{(C\,(T\,a_1), \pi_0)\} = \mathit{matches}(\pi, \Theta) \quad \Theta, \Phi_1 \vdash^{\mathtt{simp}} (C\,(T\,a_1)) \rightsquigarrow \ \mathit{fail}}{\Theta, \Phi_0 \vdash^{\mathtt{simp}} \pi \rightsquigarrow \ \mathit{fail}}$$

onde $\Phi_1 = \Phi_0[\pi, \pi_0]$, $\Phi_1(\pi_0).I = (\eta(\pi) = 1, \infty)$. Temos que:

$$\frac{\{(C\,(T^2\,a_2), \pi_0)\} = \mathit{matches}(C\,(T\,a_1), \Theta) \quad \Theta, \Phi_2 \vdash^{\mathtt{simp}} (C\,(T^2\,a_2)) \rightsquigarrow \ \mathit{fail}}{\Theta, \Phi_1 \vdash^{\mathtt{simp}} (C\,(T\,a_1)) \rightsquigarrow \ \mathit{fail}}$$

onde $\Phi_2 = \Phi_1[C\,(T\,a_1), \pi_0] = \ \mathit{Fail}$, já que $\eta(C\,(T\,a_1)) \not< \Phi_1(\pi_0).I.v_1 = 1$.

Finalmente, pela regra $(\mathtt{R}_0)$, obtemos que $\Theta \vdash^{\mathtt{simp}_0} \{D\,\mathtt{I}, C\,a\} \rightsquigarrow \{C\,a\}$, o que significa que $D\,\mathtt{I}$ pode ser removido e $C\,a$ não pode ser reduzido.

4.9 Especialização de Tipos

A especialização de tipos é um processo de simplificação de restrições que visa eliminar ambiguidades e inferir tipos mais precisos. Diversas estratégias têm sido propostas para a especialização de tipos para Haskell, como por exemplo estratégias baseadas em dependências funcionais (Seção 3.5). Nesta seção descrevemos a abordagem proposta em [Camarão et al., 2009].

Antes de apresentar essa proposta, é necessário apresentar algumas definições. Primeiramente, é apresentada a definição de [Jones, 1995b] para o conjunto de restri-

ções satisfazíveis, conceito sobre o qual a especialização de tipos é formalizada (Seção 4.9.1). Em seguida, é apresentado o fechamento de restrições (Seção 4.9.2) e finalmente, a proposta para especialização (Seção 4.9.3).

4.9.1 Conjunto de Restrições Satisfazíveis

Seguindo Mark Jones [Jones, 1995b], dado um conjunto de restrições P define-se $\lfloor P \rfloor_\Theta$ como o conjunto de instâncias de P prováveis em Θ. Mais formalmente (em que S é uma substituição qualquer):

$$\lfloor P \rfloor_\Theta = \{S\,P \mid \Theta \Vdash S\,P\}$$

Usualmente omitimos o subscrito Θ em $\lfloor P \rfloor_\Theta$, uma vez que a satisfazibilidade de P será considerada em um contexto global, contendo todas as informações de classes e instâncias presentes em um programa.

Se $\Theta \Vdash S\,P$, dizemos que S é uma substituição que satisfaz P em Θ. Para qualquer substituição S, temos que $\lfloor S\,P \rfloor \subseteq \lfloor P \rfloor$. Porém, $\lfloor P \rfloor \subseteq \lfloor S\,P \rfloor$ nem sempre é uma inclusão válida, mas permite caracterizar a especialização de P para um conjunto de restrições equivalentes $S\,P$, mais simples. Neste caso, dizemos que S é uma substituição de especialização, pois esta preserva o conjunto de restrições satisfatíziveis de P, isto é: $\lfloor P \rfloor = \lfloor S\,P \rfloor$.

Neste trabalho, definimos um algoritmo para cálculo de uma substituição de especialização, baseado na satisfazibilidade de restrições que possuam variáveis inalcançáveis. As próximas seções apresentam esse algoritmo.

4.9.2 Fechamento de Conjunto de Restrições

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

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

Figura 4.7. Fechamento de um Conjunto de Restrições

Intuitivamente, $P\,|_V^*$ representa 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 $\alpha \in tv(\pi)$, para algum $\pi \in P$ 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* em $\forall\,\overline{\alpha}.\,P \Rightarrow \tau$ se $\gamma \in tv(P) - P\,|^{*}_{(tv\,\tau)}$.

Exemplo 18. Considere o seguinte conjunto de restrições:

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

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 porque possui uma variável de tipo (`a`) que está presente em uma restrição que possui uma variável (`c`) pertencente ao conjunto $\{\texttt{c}\}$.

4.9.3 Proposta para Especialização de Tipos

Uma motivação de Mark Jones [Jones, 1995b] para a especialização de tipos é que o algoritmo de inferência pode retornar tipos menos precisos do que o esperado, por não levar em consideração a satisfazibilidade das restrições presentes nestes tipos.

Em [Camarão et al., 2009] é apresentada uma proposta de especialização baseada no conceito de variáveis inalcançáveis. Intuitivamente, um tipo $\forall\,\overline{\alpha}_1.P \Rightarrow \tau$ pode ser especializado para $\forall\,\overline{\alpha}_2.Q \Rightarrow \tau$ se existir uma única solução para a satisfazibilidade de todas as restrições de P que possuem variáveis inalcançáveis. Neste caso, o conjunto Q consiste apenas de restrições que possuem variáveis alcançáveis a partir de τ.

Essa idéia é formalizada na Figura 4.8, onde $\Theta \vdash^{\texttt{impr}} \sigma \rhd \sigma'$, denota que o tipo σ pode ser especializado para σ' utilizando informações contidas nos contexto Θ.

$$\boxed{\Theta \vdash^{\texttt{impr}} \sigma \rhd \sigma'}$$

$$\frac{\begin{array}{c}\sigma = \forall\,\overline{\alpha}\,.\,P \Rightarrow \tau \qquad P_r = P\,|^{*}_{(tv(\tau))} \quad P_u = P - P_r \\ \Theta,\,\Phi_0 \vdash^{\texttt{tsat}} P_u \rightsquigarrow \{S\} \qquad \overline{\alpha}_1 = tv(P_r \Rightarrow \tau) \\ \sigma' = \forall\,\overline{\alpha}_1\,.\,P_r \Rightarrow \tau\end{array}}{\Theta \vdash^{\texttt{impr}} \sigma \rhd \sigma'}\ \texttt{(Impr)}$$

Figura 4.8. Especialização de Tipos

Essa abordagem para especialização de tipos retarda o teste de satisfazibilidade, ao contrário do que é realizado por dependências funcionais [Jones & Diatchki, 2008].

Exemplo 19. Considere o seguinte trecho de código Haskell que poderia fazer parte de uma biblioteca de funções relacionadas a álgebra linear:

```
class Mult a b c where (.*.) :: a → b → c
instance Mult Matrix Matrix Matrix where ...
instance Mult Matrix Vector Matrix where ...

m1, m2, m3 :: Matrix
-- definição de m1, m2 e m3

m = (m1 .*. m2) .*. m3
```

Por simplicidade, denotamos os tipos `Matrix` por `M` e `Vector` por `V`. Neste caso, teríamos que $\Theta^{cls} = \{\forall$ `a b c.Mult a b c`$\}$, $\Theta^{ins} = \{$`Mult M M M`, `Mult M V M`$\}$ e o tipo inferido para `m` seria:

```
m ::  (Mult M M a, Mult a M b) ⇒ b
```

Uma vez que `m` é definido por valores do tipo `Matrix`, seria razoável conjecturar, com base nas definições de instâncias da classe `Mult`, que `m` também deve ser um valor do tipo `Matrix`.

Ao anotarmos `m::M` o compilador Haskell GHC rejeita a definição de `m`, por considerá-la ambígua (GHC considera como ambíguo todo tipo $\forall\,\overline{\alpha}\,.\,P \Rightarrow \tau$ que possui restrições contendo variáveis inalcançáveis a partir de $tv(\tau)$), pois, com esta anotação, temos:

```
m ::  (Mult M M a, Mult a M M) ⇒ M
```

Isso torna a variável de tipo `a` inalcançável e, por consequência, o GHC considera o tipo de `m` ambíguo. Porém, de acordo com a definição da Figura 4.8, o tipo

```
m ::  (Mult M M a, Mult a M M) ⇒ M
```

pode ser especializado para `m::M`, uma vez que existe uma única solução ($S = [$`a` $\mapsto$ `M`$]$) para a satisfazibilidade das restrições `Mult M M a` e `Mult a M M` em Θ.

Observe que, de acordo com a proposta baseada em variáveis inalcançáveis, a especialização (e por consequência, o teste de satisfazibilidade) só acontece quando requerido pelo contexto do programa no qual a expressão ocorre. Neste exemplo, uma anotação de tipo foi utilizada com este intuito, mas o mesmo efeito pode ser obtido por uma aplicação, conforme exemplo a seguir:

```
inverse :: Matrix →Matrix
inverse x = ...
```

```
mi = inverse m
```

Neste trecho de código, a especialização do tipo de `m` é forçada pela aplicação da função `inverse`, que possui como parâmetro um valor de tipo `Matrix`.

4.10 Definição do Sistema de Tipos

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

4.10.1 Simplificação de Tipos

As informações sobre restrições de classes e instâncias presentes no contexto Θ podem ser utilizadas para simplificar um tipo $\forall\,\overline{\alpha}\,.\,P \Rightarrow \tau$, de maneira a preservar o conjunto de instâncias que satisfazem às restrições P.

Intuitivamente, o processo de simplificação consiste em realizar a especialização de tipos com base em restrições contendo variáveis inalcançáveis e redução de contexto nas demais restrições.

Um tipo $\forall\,\overline{\alpha}\,.\,P \Rightarrow \tau$ pode ser simplificado para $\forall\,\overline{\alpha}_1\,.\,Q \Rightarrow \tau$ se $\Theta \vdash \forall\,\overline{\alpha}\,.\,P \Rightarrow \tau \gg \forall\,\overline{\alpha}_1\,.\,Q \Rightarrow \tau$ é provável de acordo com a regra definida na Figura 4.9.

$$\boxed{\Theta \vdash \forall\,\overline{\alpha}\,.\,P \Rightarrow \tau \gg \forall\,\overline{\alpha}_1\,.\,Q \Rightarrow \tau}$$

$$\frac{\begin{array}{c}\Theta \vdash^{\texttt{impr}} (\forall\,\overline{\alpha}\,.\,P \Rightarrow \tau) \rhd (\forall\,\overline{\alpha}'\,.\,P_r \Rightarrow \tau) \\ \Theta,\,\Phi_0 \vdash^{\texttt{simp}} P_r \rightsquigarrow Q \qquad\qquad \overline{\alpha}_1 = tv(Q \Rightarrow \tau)\end{array}}{\Theta \vdash \forall\,\overline{\alpha}\,.\,P \Rightarrow \tau \gg \forall\,\overline{\alpha}_1\,.\,Q \Rightarrow \tau}\ \text{(TySimp)}$$

Figura 4.9. Simplificação de Tipos.

A definição de simplificação de tipos será utilizada na função de generalização de tipos (Seção 4.10.2).

4.10.2 O Sistema de Tipos

Um sistema de tipos dirigido por sintaxe para *core-Haskell* é apresentado na Figura 4.10. As regras são similares as regras do sistema de tipos de Hindley-Milner [Milner, 1978], exceto pelo uso da instanciação de tipos restringidos (Seção 4.6.2) na

regra (VAR). Na regra (LET) é feita a simplificação do tipo inferido (através da função de generalização), antes que este seja incluído em Γ.

A notação $\Gamma\,[x : \sigma]$ representa a operação de incluir a suposição $x : \sigma$ em Γ, isto é, $\Gamma\,[x : \sigma] = (\Gamma - \{\Gamma(x)\}) \cup \{x : \sigma\}$.

A função *gen* realiza a generalização de um tipo $P \Rightarrow \tau$, simplificando-o de acordo com a definição de simplificação de tipos (Figura 4.9).

$$
\begin{array}{rcl}
gen(\Theta, \Gamma, P \Rightarrow \tau) & = & \forall\, \overline{\alpha}\,.\, Q \Rightarrow \tau \\
 & & \text{onde:} \\
 & & \quad \Theta \vdash P \Rightarrow \tau \gg Q \Rightarrow \tau \\
 & & \quad \overline{\alpha} = tv(Q \Rightarrow \tau) - tv(\Gamma)
\end{array}
$$

$$
\boxed{\Theta \mid \Gamma \vdash e : P \Rightarrow \tau}
$$

$$
\frac{\Gamma(x) = \sigma \quad \sigma \leq_{\Sigma} P \Rightarrow \tau}{\Theta \mid \Gamma \vdash x : P \Rightarrow \tau}\ \text{(VAR)}
$$

$$
\frac{\Theta \mid \Gamma\,[x : \tau_1] \vdash e : P \Rightarrow \tau_2}{\Theta \mid \Gamma \vdash \lambda\, x\,.\, e : P \Rightarrow \tau_1 \to \tau_2}\ \text{(ABS)}
$$

$$
\frac{\Theta \mid \Gamma \vdash e_1 : P_1 \Rightarrow \tau_2 \to \tau_1 \quad \Theta \mid \Gamma \vdash e_1 : P_2 \Rightarrow \tau_2}{\Theta \mid \Gamma \vdash e_1\, e_2 : (P_1 \cup P_2) \Rightarrow \tau_1}\ \text{(APP)}
$$

$$
\frac{\Theta \mid \Gamma \vdash e_1 : P_1 \Rightarrow \tau_1 \quad \sigma_1 = gen(\Theta, \Gamma, P_1 \Rightarrow \tau_1) \quad \Theta \mid \Gamma\,[x : \sigma_1] \vdash e_2 : P \Rightarrow \tau}{\Theta \mid \Gamma \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : P \Rightarrow \tau}\ \text{(LET)}
$$

Figura 4.10. Regras do Sistema de Tipos

4.11 Inferência de Tipos

O algoritmo de inferência de tipos é apresentado na Figura 4.11, como um sistema de provas dirigido por sintaxe, para julgamentos da forma $\Theta \mid \Gamma \vdash_{\mathtt{I}} e : (P \Rightarrow \tau, S)$, em que Θ e Γ são, respectivamente, os contextos contendo informações sobre restrições de classes, instâncias e suposições de tipos, e é uma expressão, $P \Rightarrow \tau$ é o tipo inferido para essa expressão, e S é uma substituição.

A notação $P \oplus_V Q$ denota o conjunto de restrições obtido por adicionar apenas restrições de Q que possuam variáveis alcançáveis a partir de V, isto é: $P \oplus_V Q =$

$P \cup \left(Q\,|^*_V\right)$. Eliminar restrições contendo variáveis inalcançáveis de Q é necessário uma vez que estas podem se referir a "partes" não selecionadas do argumento. Como exemplo, considere a expressão $e = \mathit{fst}(\texttt{True}, o)$, onde o possui um conjunto não vazio de restrições. O tipo de e deve ser `Bool`, já que o valor desta expressão não depende de o e, portanto, restrições presentes no tipo de o não devem ser incluídas no tipo de e.

$$\boxed{\Theta \mid \Gamma \vdash_{\mathtt{I}} e : (P \Rightarrow \tau, S)}$$

$$\frac{\Gamma(x) = \forall\, \overline{\alpha}\,.P \Rightarrow \tau \quad \overline{\beta} \text{ são novas variáveis}}{\Theta \mid \Gamma \vdash_{\mathtt{I}} x : (\,[\overline{\beta} \mapsto \overline{\alpha}]\,P \Rightarrow \tau, id)}\ (\mathrm{VAR}_{\mathtt{I}})$$

$$\frac{\Theta \mid \Gamma\,[x : \alpha] \vdash_{\mathtt{I}} e : (P \Rightarrow \tau, S) \quad \tau' = S\,\alpha}{\Theta \mid \Gamma \vdash_{\mathtt{I}} \lambda x\,.\,e : (P \Rightarrow \tau' \to \tau, S)}\ (\mathrm{ABS}_{\mathtt{I}})$$

$$\frac{\begin{array}{cc}\Theta \mid \Gamma \vdash_{\mathtt{I}} e_1 : (P_1 \Rightarrow \tau_1, S_1) & \alpha \text{ é uma nova variável} \\ \Theta \mid S_1\,\Gamma \vdash_{\mathtt{I}} e_2 : (P_2 \Rightarrow \tau_2, S_2) & S' = \mathit{mgu}(\{\tau_1 = \tau_2 \to \alpha\}) \\ \tau = S\,\alpha \quad V = tv(\tau) & P = P_1 \oplus_V P_2 \quad S = S' \circ S_2 \circ S_1\end{array}}{\Theta \mid \Gamma \vdash_{\mathtt{I}} e_1\,e_2 : (P \Rightarrow \tau, S)}\ (\mathrm{APP}_{\mathtt{I}})$$

$$\frac{\Theta \mid \Gamma \vdash_{\mathtt{I}} e_1 : (Q \Rightarrow \tau', S') \quad \sigma = \mathit{gen}(\Theta, \Gamma, Q \Rightarrow \tau') \quad \Theta \mid \Gamma\,[x : \sigma] \vdash_{\mathtt{I}} e_2 : (P \Rightarrow \tau, S)}{\Theta \mid \Gamma \vdash_{\mathtt{I}} \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 : (P \Rightarrow \tau, S)}\ (\mathrm{LET}_{\mathtt{I}})$$

Figura 4.11. Algoritmo de Inferência de Tipos

O algoritmo de inferência da Figura 4.11 é similar ao algoritmo apresentado por M. Jones em [Jones, 1995a], exceto pelo fato de que a especialização de tipos é realizada na regra ($\mathrm{LET}_{\mathtt{I}}$).

Propriedades de terminação e corretude do algoritmo proposto em relação ao sistema de tipos são enunciadas e provadas no Apêndice A.

4.11.1 Incompletude e Ambiguidade

O leitor atento deve ter notado as similaridades entre o algoritmo de inferência da Figura 4.11 e o sistema de tipos da Figura 4.10. Tal semelhança induz as seguintes questões sobre o algoritmo de inferência em relação ao sistema de tipos:

1. Se o algoritmo encontra um tipo $P \Rightarrow \tau$ para uma expressão e em contextos $\Theta \mid \Gamma$, existe uma derivação no sistema de tipos tal que $\Theta \,|\, \Gamma \vdash e : P \Rightarrow \tau$?

2. Se existe uma derivação de $\Theta \,|\, \Gamma \vdash e : P \Rightarrow \tau$ no sistema de tipos, o algoritmo proposto encontra o tipo $P \Rightarrow \tau$ para e a partir das informações contidas nos contextos Θ e Γ ?

A primeira pergunta remete à propriedade de **correção** de um algoritmo de inferência de tipos, isto é, se o algoritmo calcula um determinado tipo para uma expressão, então o tipo calculado é derivável utilizando a especificação do sistema de tipos. A segunda pergunta lida com a questão da **completude** de um algoritmo. Diz-se que um algoritmo de inferência é completo se este é capaz de calcular tipos para toda expressão que possui um tipo derivável utilizando o sistema de tipos.

A correção do algoritmo de inferência da Figura 4.11 é facilmente demonstrada por indução sobre a derivação de $\Theta \mid \Gamma \vdash_{\mathtt{I}} e : (P \Rightarrow \tau, S)$. Já a completude não é válida em geral, devido á indecidibilidade da satisfazibilidade de restrições e á possibilidade de derivar no sistema de tipos da Figura 4.10, tipos para expressões que o algoritmo de inferência da Figura 4.11 reporta como ambíguos. Para solucionar o problema da satisfazibilidade de restrições, consideraremos uma versão restrita deste problema. Definimos que um conjunto de restrições P é considerado satisfazível se $\Theta, \Phi \vdash^{\mathtt{tsat}} P \rightsquigarrow \mathbb{S}$ (Figura 4.5) é provável. O problema da ambiguidade é relacionado ao fato e que instanciação de tipos é independente de contexto. O próximo exemplo ilustra este problema.

Exemplo 20. Considere o seguinte trecho de código:

```
class Show a where show ::  a → String
class Read a where read ::  String → a

instance Show Int where ...
instance Show Bool where ...

instance Read Int where ...
instance Read Bool where ...

echo x = show (read x)
```

O tipo inferido para `echo` é:

```
echo ::  (Show α, Read α) ⇒ String → String
```

Esse tipo é rejeitado pelo algoritmo de inferência, pois não existe nesse contexto uma solução única para a satisfazibilidade de $P = \{\texttt{Show } \alpha, \texttt{ Read } \alpha\}$. Porém, `echo` é tipável de acordo com as regras do sistema de tipos.

Apesar de aceitos pela especificação, tais programas devem ser rejeitados, porque definições com tipos ambíguos podem fazer com que o significado de um programa seja afetado por escolhas realizadas pelo algoritmo de inferência de tipos. Uma possível solução para este problema seria modificar o sistema de tipos, para que ele não permita a derivação de tipos ambíguos, tal como propomos na seção a seguir.

4.12 Solucionando o Problema de Incompletude

Sistemas de tipos tal como o apresentado na Figura 4.10, que provêem suporte para sobrecarga dependente de contexto e seguem a abordagem de Hindley-Milner de instanciação livre de contexto, permitem derivações distintas do mesmo tipo para algumas expressões, que então são consideradas ambíguas. Tais expressões são usualmente rejeitadas por algoritmos de inferência de tipos e por isso tais algoritmos não são completos com respeito ao sistema de tipos em questão.

A incompletude de algoritmos de inferência de tipos e problemas para a definição de uma semântica coerente são discutidos, por exemplo, em [Vytiniotis et al., 2011]. K. Faxén [Faxén, 2002] cita que, idealmente, deveria haver uma maneira determinista e concisa para não permitir que seja possível construir derivações distintas de um mesmo tipo para uma expressão, utilizando a especificação do sistema de tipos.

Para exemplificar a relação entre instanciações de tipos e a incompletude do algoritmo de inferência apresentado na Figura 4.11, considere novamente a definição da função `echo` apresentada no Exemplo 20:

```
echo x = show (read x)
```

onde Γ possui os tipos das funções `read` e `show` e Θ possui as restrições correspondentes às definições de classes e instâncias do Exemplo 20. O tipo de `echo` é:

$$\texttt{echo :: (Show } \alpha\texttt{, Read } \alpha\texttt{)} \Rightarrow \texttt{String} \rightarrow \texttt{String}$$

O uso de `echo` é rejeitado pelo algoritmo da Figura 4.11 por não existir uma única solução para satisfazibilidade das restrições $\texttt{(Show } \alpha\texttt{, Read } \alpha\texttt{)}$ em Θ. Porém, é possível construir derivações de tipo para `echo`, utilizando as regras do sistema de tipos da Figura 4.10, instanciando a variável de tipos α para `I` ou para `B`. A Figura 4.12

apresenta um trecho da derivação onde variável α é instanciada para o tipo `B` (nesta figura, abreviamos `Show` e `Read` para `S` e `R` respectivamente).

O sistema de tipos da Figura 4.10 permite derivações para expressões ambíguas, como a derivação da Figura 4.12, porque possibilita instanciação de tipo na regra `(VAR)`, a qual reproduzimos abaixo por conveniência:

$$\frac{\Gamma(x) = \sigma \quad \sigma \leq_\Sigma P \Rightarrow \tau}{\Theta \mid \Gamma \vdash x : P \Rightarrow \tau} \text{ (VAR)}$$

A condição $\sigma \leq_\Sigma P \Rightarrow \tau$ permite que um tipo σ seja instanciado de maneira não determinista, para qualquer tipo $P \Rightarrow \tau$ que satisfaça a ordenação de tipos definida na Seção 4.6.2. Com isso, é possível construir derivações para expressões que possuem tipos ambíguos, tal como a que define a função `echo`. Para solucionar este problema, propomos definir um sistema de tipos que permita apenas instanciações dependentes de contexto, de maneira que uma expressão, caso seja tipável, possua um único tipo (módulo renomeação de variáveis).

4.12.1 Sistema de Tipos Para Instanciações Dependentes de Contexto

A Figura 4.13 apresenta um sistema de tipos para core Haskell que não permite instanciações independentes de contexto. O sistema de tipos é apresentado como um conjunto de regras para derivar julgamentos da forma $\Theta \mid \Gamma \vdash^D e : (P \Rightarrow \tau, S)$, que denota que e possui o tipo $P \Rightarrow \tau$ no contexto $\Theta \mid \Gamma$ e S é uma substituição tal que $\Theta \mid S\,\Gamma \vdash^D e : (P \Rightarrow \tau, S')$ é provável e $S' \leq_\mathbb{S} S$.

Na Figura 4.13, a notação $\tau \overset{S}{\approx} \tau'$ representa a relação de triplas formadas por dois tipos τ, τ' e uma substituição S tal que S é o unificador mais geral de τ e τ' (caso ele exista).

Sempre que $\Theta \mid \Gamma \vdash^D e : (P \Rightarrow \tau, S)$ é provável, a substituição S pode ser utilizada para instanciar os tipos em Γ de variáveis livres presentes em e, obtendo uma nova derivação $\Theta \mid S\,\Gamma \vdash^D e : (P \Rightarrow \tau, S')$ onde S' é mais geral que S. O próximo exemplo ilustra este fato.

Exemplo 21. Considere a expressão x, $\Gamma = \{f : Int \to Int, x : \alpha\}$ e Θ um contexto vazio de informações sobre símbolos sobrecarregados. Podemos derivar $\Theta \mid \Gamma \vdash^D f\,x : (Int, S)$, onde $S = [\alpha \mapsto Int]$. A partir de $S\,\Gamma = \{f : Int \to Int, x : Int\}$; podemos derivar $\Theta \mid S\,\Gamma \vdash^D f\,x : (Int, id)$ e, evidentemente, temos que $id \leq_\mathbb{S} S$.

Informalmente, dizemos que a instanciação de um tipo σ de uma expressão e é dependente de contexto, se esta somente ocorre quando exigido pelo ponto do programa onde a expressão e ocorre, isto é, somente na aplicação de uma função a seus argumentos. Formalmente, o contexto de uma expressão e, $\mathcal{C}[e]$, é uma expressão e', $e' \neq e$, tal que e é uma subexpressão de e'. A relação entre as derivações de tipo para e e $\mathcal{C}[e]$ é expressa pelo Teorema 13 (presente na Seção A.5 do Apêndice A), cujo enunciado é o seguinte.

Teorema: Se $\Theta \mid \Gamma \vdash^D e : (P \Rightarrow \tau, S)$ então, em todos os contextos $\mathcal{C}[e]$ e todo Γ' tal que $\Gamma \leq_\omega \Gamma'$ e $\Theta \mid \Gamma' \vdash^D \mathcal{C}[e] : (P' \Rightarrow \tau', S')$ é provável para algum $P' \Rightarrow \tau'$ e S' temos que $S \leq_{\mathbb{S}} S'$.

Para cada expressão e, se e é tipável em um contexto $\Theta \mid \Gamma$, então existe um único tipo $P \Rightarrow \tau$ derivável para e neste contexto. Todavia, o tipo de e pode ser entendido como um conjunto de instâncias, em contextos de programa que exigem a instanciação de $P \Rightarrow \tau$ em Γ (cf. Teorema 13). Isso é ilustrado pelo exemplo a seguir em que `B` e `C` abreviam `Bool` e `Char`, respectivamente.

Exemplo 22. Seja $\Gamma = \{\texttt{(==)} : \forall a.\, Eq\ a \Rightarrow a \to a \to \texttt{B}\}$, $\Theta^{cls} = \{Eq\ a\}$, $\Theta^{ins} = \{Eq\ \texttt{B}, Eq\ \texttt{C}\}$ e $e_1 =$ `((==) True, (==) '*')`. Então, $\Theta \mid \Gamma \vdash^D e_1 : ((\texttt{B} \to \texttt{B}, \texttt{C} \to \texttt{B}), S)$ é derivável, onde $S = [a \mapsto \texttt{B}, b \mapsto \texttt{C}]$, e a, b são novas variáveis. Note que os tipos inferidos para cada ocorrência de `(==)` são instanciados nos contextos `(==) True` e `(==) '*'` para $\texttt{B} \to \texttt{B} \to \texttt{B}$ e $\texttt{C} \to \texttt{C} \to \texttt{B}$, respectivamente.

Instâncias de tipos são definidas formalmente da seguinte forma. Dada uma expressão e e contextos Θ e Γ tais que $\Theta \mid \Gamma \vdash^D e : (P \Rightarrow \tau, S)$, temos que $S'(P \Rightarrow \tau)$ é uma instância do tipo de e em Θ e Γ se $\Theta \mid \Gamma' \vdash^D \mathcal{C}[e] : (P' \Rightarrow \tau', S')$ é provável para algum Γ' tal que $\Gamma \leq_\omega \Gamma'$. Além disso, $S'(P \Rightarrow \tau)$ é a maior (mais específica) instância de tipo para uma ocorrência de e em Θ e Γ se $S'(P \Rightarrow \tau)$ é uma instância do tipo de e e não existe uma instância $S_1(P \Rightarrow \tau)$ para e em $\Theta \mid \Gamma$ distinta de $S'(P \Rightarrow \tau)$, tal que $S_1 \leq_{\mathbb{S}} S'$.

Ocorrências distintas de uma expressão podem possuir diferentes maiores instâncias do tipo desta expressão. No Exemplo 22, as duas ocorrências de `(==)` possuem duas diferentes instâncias do tipo desta função. O tipo de cada uma dessas ocorrências é a maior instância do tipo de `(==)` em seu respectivo contexto.

O sistema de tipos da Figura 4.13 difere do apresentado na Figura 4.10 por não permitir a instanciação de tipos na regra para variáveis. O algoritmo de inferência da Figura 4.11 é correto e completo com respeito ao sistema de tipos da Figura 4.13. Esta prova é apresentada no Apêndice A.

4.13 Semântica

Classes de tipos definem símbolos sobrecarregados, usualmente denominados membros da classe, com seus respectivos tipos e uma declaração de instância fornece uma definição para cada membro da classe.

A semântica para core Haskell, apresentada na Figura 4.14, é baseada em um esquema de tradução em que símbolos sobrecarregados são transformados em aplicações dos respectivos *dicionários* das classes, conforme definido em [Wadler & Blott, 1989, Jones, 1995a, Hall et al., 1996]. Um dicionário de classe consiste de uma tupla, correspondente a uma declaração de instância, que contém cada uma das definições dos membros da classe em questão. Além disso, dicionários devem conter referências para dicionários de super-classes. Neste trabalho não vamos considerar o tratamento de super-classes, uma vez que pode ser feito conforme as abordagens previamente descritas na literatura [Jones, 1995a, Hall et al., 1996, Faxén, 2002].

A Figura 4.14 define a semântica para core Haskell por indução sobre as regras do sistema de tipos da Figura 4.13, onde variáveis estão anotadas com a maior instância de seu tipo no contexto onde estas ocorrem. A notação $x :: (P \Rightarrow \tau)$ indica que a variável x possui como maior instância o tipo $(P \Rightarrow \tau)$. A semântica da Figura 4.14 utiliza um ambiente η, que armazena informações para a tradução de símbolos sobrecarregados utilizando dicionários.

Cada definição de classe

$$\texttt{class}\ P \Rightarrow C\,\overline{\alpha}\ \texttt{where}\ \overline{x} :: \overline{\tau}$$

define um conjunto de funções de projeção para dicionários da classe C. A função de projeção para o símbolo x_i retorna o i-ésimo componente da tupla que define um dicionário da classe C (se $n = 1$, a projeção é feita pela função identidade).

Denotamos por $\overline{P}$ uma sequência, em ordem lexicográfica, de restrições $\pi \in P$.

Para cada definição de instância

$$\texttt{instance}\ P \Rightarrow C\,\overline{\mu}\ \texttt{where}\ \overline{x} = \overline{e}$$

é criado um dicionário d_π, da classe C. Cada componente de d_π é uma função que recebe como parâmetro um dicionário para cada restrição na (possivelmente vazia) sequência $\overline{P}$ e produz uma tradução de e_i, a expressão associada a x_i na declaração de instância. Cada dicionário d_π é tal que $\eta(S\,(C\,\overline{\mu})) = d_\pi$ e $\eta(x_i, S\,\tau_i) = d_\pi$, para qualquer substituição S, e τ_i é o tipo simples do tipo de x_i. A notação $\eta \dagger (P \mapsto \overline{v})$ representa $\eta\,[\pi_1 \mapsto v_1 \,...\, \pi_n \mapsto v_n]$ e $vSeq(\overline{P})$ denota uma sequência de novas variáveis, uma para cada $\pi_i \in P$, onde $P = \{\pi_1, ..., \pi_n\}$.

A notação $\eta(x, P \Rightarrow \tau, \Gamma)$ define a semântica de um nome possivelmente sobrecarregado x, com tipo $P \Rightarrow \tau$. A definição de η utiliza um pequeno abuso de notação, ao usar η sobre restrições (como em $\eta(S\,\pi)$) e para produzir dicionários (como em $\eta(x_i, S\,\tau_i)$):

$$\eta(x, P \Rightarrow \tau, \Gamma) = \begin{cases} x & \text{se } P_0 = \emptyset \\ x\;\eta(x,\tau)\;\overline{w} & \text{caso contrário.} \end{cases}$$

onde: $\forall\,\overline{\alpha}.\, P_0 \Rightarrow \tau_0 = \Gamma(x)$,
$S = mgu(\tau, \tau_0)$,
$\pi_1 \ldots \pi_n = \overline{P_0}$,
para $i = 1, \ldots, n : v_i = \eta(\pi_i)$,

$$w_i = \begin{cases} v_i & \text{se } \pi_i \in P \\ \eta(S\pi_i) & \text{caso contrário} \end{cases}$$

O seguinte exemplo ilustra a semântica proposta.

Exemplo 23. Considere o seguinte trecho de programa Haskell.

```
class TEq a where
    teq :: a → a → (Bool,String)

instance TEq Int where
    teq i i' = (i == i', show i ++ '' '' ++ show i')

instance (TEq a, Show a) ⇒ TEq [a] where
    teq [] [] = (True, '' '')
    teq (a:x) (b:y) = let (ab,sab) = teq a b
                          (xy,sxy) = teq x y
                      in (ab && xy, sab ++ sxy)
    teq _ _ = (False, '' '')
teqww x = (teq [[x]], teq ([1,2,3] ::  [Int])) -- (1)
```

A tradução da primeira ocorrência de *teq* na linha (1) acima é igual a $teq\,d_{TEqL}v_1v_2$, onde *teq* representa a função identidade, $\mathbf{teq}_L$ é uma função que recebe dois dicionários, v_1 e v_2, fornecidos como argumentos para *teqww*, e produz a tradução da função *teq* para listas, definida acima. A tradução é feita com respeito a $\eta_0 \dagger (P \mapsto \overline{v})$, onde $P = \{Show\ a, TEq\ a\}$, $\overline{v}$ é a sequência $v_1\ v_2$, e η_0 é tal que $\eta_0(teq, \tau) = d_{TEqL}$, onde $\tau = [a] \to [a] \to (Bool, String)$, e $\mathtt{d}_{TEqL}$ é um dicionário com apenas um componente $\mathbf{teq}_L$. Além disso, temos que $\eta_0(TEq\ Int)$ é igual ao dicionário com apenas um

componente (por exemplo, d_{TEqInt}), e, de maneira similar, para $\eta_0(Show\ Int)$. Sendo assim, a tradução da segunda ocorrência de *teq* na linha (1) é igual a:

$$teq\, d_{TEqL}\, d_{TEqInt}\, d_{ShowInt}$$

4.14 Aspectos de Implementação

Um protótipo do algoritmo de inferência proposto neste capítulo (Figura 4.11) foi implementado em Haskell. Este foi submetido a diversos testes incluindo bibliotecas, exemplos encontrados na literatura e casos de teste do compilador GHC, envolvendo extensões pertinentes. Dentre as bibliotecas testadas, destaca-se a biblioteca de transformadores monádicos, que faz uso extensivo de dependências funcionais.

O protótipo implementado pode ser obtido no seguinte endereço eletrônico:

http://github.com/rodrigogribeiro/mptc

A implementação é formada por 48 módulos totalizando 5222 linhas de código e possui a seguinte estrutura de pastas:

- /src/BuiltIn: Contém módulos possuindo definições de operações primitivas, utilizadas para definições das bibliotecas testadas.
- /src/Iface: Módulos responsáveis pela manipulação de arquivos de interface de módulos previamente compilados.
- /src/Libs: Bibliotecas, programas de exemplo presentes na literatura e casos de teste que foram verificados utilizando o front-end implementado.
- /src/Tc: Implementação do algoritmo de inferência e verificação de tipos.
- /src/Tc/Kc: Implementação do algoritmo de inferência de *kinds*.
- /src/Tests: Testes de unidade.
- /src/Utils: Utilidades diversas: algoritmos para análise de dependências, mônadas, operações sobre identificadores, e outras funções utilizadas por diversos módulos.

4.15 Conclusão

Neste capítulo foi apresentada a proposta deste trabalho para a adoção de classes de tipos com múltiplos parâmetros em Haskell, sem a necessidade de quaisquer extensões. Foram apresentados algoritmos corretos e que terminam para qualquer entrada para a satisfazibilidade e redução de contexto.

Provas de corretude e terminação dos algoritmos apresentados neste capítulo estão no Apêndice A.

Além disso caracterizamos o problema de incompletude do algoritmo de inferência em relação ao sistema de tipos proposto, apresentando uma definição alternativa do sistema de tipos para o qual o algoritmo de inferência proposto é correto e completo. Finalmente, apresentamos uma semântica de core Haskell, definida por indução sobre as derivações do sistema de tipos. No Apêndice A são demonstrados teoremas sobre o sistema de tipos, algoritmo de inferência e a semântica propostos.

$$\dfrac{\dfrac{\dfrac{\Gamma(\texttt{show}) = \forall\,\alpha\,.\,\texttt{S}\ \ \alpha \Rightarrow \alpha \to \texttt{String}}{\forall\,\alpha\,.\,\texttt{S}\ \ \alpha \Rightarrow \alpha \to \texttt{String} \leq_{\Sigma} \texttt{B}\ \to\ \texttt{String}}\ (VAR)}{\Theta\ \mid\ \Gamma\,[x : \texttt{B}] \vdash \texttt{show}\ :\ \ \texttt{B}\ \to\ \texttt{String}} \qquad \dfrac{\dfrac{\Gamma(\texttt{read}) = \forall\,\alpha\,.\,\texttt{R}\ \ \alpha \Rightarrow \alpha \to \texttt{String}}{\forall\,\alpha\,.\,\texttt{R}\ \ \alpha \Rightarrow \texttt{String} \to \alpha \leq_{\Sigma} \texttt{String}\ \to\ \texttt{B}}\ (VAR)}{\Theta\ \mid\ \Gamma\,[x : \texttt{B}] \vdash \texttt{read}\ :\ \ \texttt{String}\ \to\ \texttt{B}} \qquad \dfrac{\dfrac{\Gamma(x) = \texttt{B} \quad \texttt{B} \leq_{\Sigma} \texttt{B}}{\Theta\ \mid\ \Gamma\,[x : \texttt{B}] \vdash \texttt{x}\ :\ \ \texttt{B}}\ (VAR)}{\Theta\ \mid\ \Gamma\,[x : \texttt{B}] \vdash \texttt{read x}\ :\ \ \texttt{B}}\ (APP)}{\dfrac{\Theta\ \mid\ \Gamma\,[x : \texttt{B}] \vdash \texttt{show (read x)}\ :\ \ \texttt{String}}{\Theta\ \mid\ \Gamma \vdash \lambda\,x.\,\texttt{show (read x)} : \text{String} \to \text{String}}\ (ABS)}\ (APP)}{\Theta\ \mid\ \Gamma \vdash \texttt{let echo = } \lambda\,x.\,\texttt{show (read x) in echo:String}\to\texttt{String}}\ (LET)$$

Figura 4.12. Exemplo de derivação

$$\boxed{\Theta \mid \Gamma \vdash^D e : (P \Rightarrow \tau, S)}$$

$$\frac{\Gamma(x) = \forall\, \overline{\alpha}.\, P \Rightarrow \tau \quad \overline{\beta} \text{ são novas variáveis}}{\Theta \mid \Gamma \vdash^D x : (\overline{[\alpha \mapsto \beta]}\, P \Rightarrow \tau, id)}\ (VAR)$$

$$\frac{\Theta \mid \Gamma\,[x : \alpha] \vdash^D (P \Rightarrow \tau', S) \quad \alpha \text{ é uma nova variável} \quad \tau = S\,\alpha}{\Theta \mid \Gamma \vdash^D \lambda x.e : (P \Rightarrow \tau \to \tau', S)}\ (ABS)$$

$$\frac{\begin{array}{c}\Theta \mid \Gamma \vdash^D e_1 : (P_1 \Rightarrow \tau_1, S_1) \quad \Theta \mid S_1\,\Gamma \vdash^D e_2 : (P_2 \Rightarrow \tau_2, S_2) \\ S_2\,\tau_1 \overset{S'}{\approx} \tau_2 \to \alpha \qquad \alpha \text{ é uma nova variável} \quad S = S' \circ S_2 \circ S_1 \\ \tau = S\,\alpha \quad V = tv(\tau) \quad P = S\,P_1 \oplus_V S\,P_2 \quad \Theta \vdash P \Rightarrow \tau \gg Q \Rightarrow \tau\end{array}}{\Theta \mid \Gamma \vdash^D e_1\,e_2 : (Q \Rightarrow \tau, S)}\ (APP)$$

$$\frac{\begin{array}{c}\Theta \mid \Gamma \vdash^D e_1 : (P_1 \Rightarrow \tau_1, S_1) \\ \overline{\alpha} = tv(P_1 \Rightarrow \tau_1) - tv(\Gamma) \\ \sigma = \forall\, \overline{\alpha}.P_1 \Rightarrow \tau_1 \qquad \Theta \mid \Gamma\,[x : \sigma] \vdash^D e_2 : (P \Rightarrow \tau, S)\end{array}}{\Theta \mid \Gamma \vdash^D \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : (P \Rightarrow \tau, S)}\ (LET)$$

Figura 4.13. Sistema de tipos para instanciações dependentes de contexto

$$\boxed{[\![\Theta \mid \Gamma \vdash^D e : (P \Rightarrow \tau, S)]\!]_\eta = \mathbf{e} : \sigma}$$

$$\frac{\Gamma(x) = \sigma}{[\![\Theta \mid \Gamma \vdash^D (x :: P \Rightarrow \tau) : (P' \Rightarrow \tau', S')]\!]_\eta = \eta(x, P \Rightarrow \tau, \Gamma) : \tau} \; (VAR)$$

$$\frac{[\![\Theta \mid \Gamma\,[x : \alpha] \vdash^D e : (P \Rightarrow \tau, S)]\!]_\eta = \mathbf{e} : \tau \quad \alpha \text{ é uma nova variável} \quad \tau' = S\,\alpha}{[\![\Theta \mid \Gamma \vdash^D \lambda x.e : (P \Rightarrow \tau' \to \tau, S)]\!]_\eta = \lambda x.\mathbf{e} : \tau' \to \tau} \; (ABS)$$

$$\frac{\begin{array}{c}[\![\Theta \mid \Gamma \vdash^D e_1 : (P_1 \Rightarrow \tau_1, S_1)]\!]_\eta = \mathbf{e}_1 : S\tau_1 \\ [\![\Theta \mid \Gamma \vdash^D e_2 : (P_2 \Rightarrow \tau_2, S_2)]\!]_\eta = \mathbf{e}_2 : S\tau_2 \\ S' = mgu(\{S_2\tau_1 = \tau_2 \to \alpha\}) \quad \alpha \text{ é uma nova variável.} \\ S = S' \circ S_2 \circ S_1 \qquad \tau = S\,\alpha \qquad V = tv(\tau) \\ P = S\,P_1 \oplus_V S\,P_2 \qquad \Theta \vdash P \Rightarrow \tau \gg Q \Rightarrow \tau\end{array}}{[\![\Theta \mid \Gamma \vdash^D e_1\,e_2 : (Q \Rightarrow \tau, S)]\!]_\eta = \mathbf{e}_1\,\mathbf{e}_2 : \tau} \; (APP)$$

$$\frac{\begin{array}{cc}[\![\Theta \mid \Gamma \vdash^D e_1 : (P_1 \Rightarrow \tau_1, S_1)]\!]_\eta = \mathbf{e}_1 : \tau_1 & [\![\Theta \mid \Gamma\,[x : \sigma] \vdash^D e_2 : (P_2 \Rightarrow \tau_2, S)]\!]_{\eta'} = \mathbf{e}_2 : \tau_2 \\ \sigma = gen(P_1 \Rightarrow \tau_1, tv(S_1\,\Gamma)) & \sigma' = gen(P_1 \Rightarrow \tau_1, tv(S\,(S_1\,\Gamma))) \\ \overline{v} = vSeq(P_1) & \eta' = \eta \dagger (P_1 \mapsto \overline{v})\end{array}}{[\![\Theta \mid \Gamma \vdash^D \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : (P \Rightarrow \tau, S)]\!]_\eta = \mathbf{let}\ x = \lambda\overline{v}.\mathbf{e}_1\ \mathbf{in}\ \mathbf{e}_2 : \tau} \; (LET)$$

Figura 4.14. Semântica para Core-Haskell

Capítulo 5

Classes de Tipos Opcionais em Haskell

Neste capítulo apresentamos uma alternativa para a definição opcional de classes de tipos em Haskell. Na Seção 5.1 são apresentadas motivações para definição opcional de classes de tipos, a Seção 5.2 define o algoritmo para cálculo da generalização de tipos simples e a Seção 5.3 apresenta a formalização da proposta para classes opcionais.

5.1 Motivação

Classes de tipos são utilizadas em Haskell para definir o tipo principal de símbolos sobrecarregados. Toda vez que o desenvolvedor deseja definir uma função sobrecarregada, é necessário que ele declare uma classe e o respectivo tipo de cada símbolo sobrecarregado da classe [Jones, 2002]. Para definição de uma classe, o programador deve determinar, a priori, quais símbolos devem ser membros desta classe. Entretanto, a questão do agrupamento de símbolos relacionados em uma construção de uma linguagem de programação deve estar relacionada ao desenvolvimento modular de software, e não à definição de símbolos sobrecarregados [Camarão & Figueiredo, 1999].

Para cada nova definição de um símbolo sobrecarregado, o tipo desta definição deve ser uma instância do tipo principal anotado na definição da classe. Se este não for o caso, a declaração da classe deve ser alterada, para que a nova definição possa ser incluída.

Visando solucionar esses problemas, este trabalho descreve uma alternativa em que o desenvolvedor pode fazer as definições sobrecarregadas que desejar, sendo o cálculo do tipo destes símbolos determinado automaticamente. Caso uma nova definição

seja feita e ela não seja instância desse tipo, o tipo é recalculado para refletir a nova definição. Visando ilustrar esta abordagem, considere os exemplos a seguir.

Exemplo 24. Considere a função `map` que opera sobre listas:

```
map f [] = []
map f (x:xs) = f x : map f xs
```

O tipo inferido para essa definição é:

```
(a → b) → [a] → [b]
```

Uma função similar pode ser definida, por exemplo, para árvores binárias, como a seguir:

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


map f Leaf = Leaf
map f (Node v l r) = Node (f v) (map f l) (map f r)
```

O tipo desta implementação de `map` é

```
(a → b) → Tree a → Tree b
```

Para determinar o tipo de `map` em um contexto contendo essas duas definições, basta calcular a menor generalização comum (Seção 5.2) dos tipos simples destas definições. A menor generalização de

$$\texttt{(a} \rightarrow \texttt{b)} \rightarrow \texttt{[a]} \rightarrow \texttt{[b]}$$

e

$$\texttt{(a} \rightarrow \texttt{b)} \rightarrow \texttt{Tree a} \rightarrow \texttt{Tree b}$$

é o tipo simples

$$\texttt{(a} \rightarrow \texttt{b)} \rightarrow c\ \texttt{a} \rightarrow c\ \texttt{b}.$$

onde c é uma nova variável de tipo, introduzida para generalizar os construtores de tipos de listas e de árvores. O tipo de `map` é obtido criando uma restrição, correspondente a uma classe que contém esta função como único membro, isto é, é gerada uma restrição contendo as variáveis de tipos que foram introduzidas por generalizações. Sendo assim, é gerada a restrição `Map` c, a partir do resultado da menor generalização dos tipos de `map`.

Finalmente, o tipo calculado para `map` é:

$$\texttt{map::(Map c)} \Rightarrow \texttt{(a} \rightarrow \texttt{b)} \rightarrow \texttt{c a} \rightarrow \texttt{c b}$$

É necessário criar automaticamente, no contexto Θ, classes e instâncias que refletem as definições de `map`, ou seja: $\Theta^{cls}(\texttt{Map}) = \forall\, c\,.\,\texttt{Map}\ c$ e $\Theta^{ins}(\texttt{Map}) = \{\texttt{Map []}, \texttt{Map Tree}\}$.

O próximo exemplo mostra a inclusão de uma nova definição da função `map` cujo tipo não é uma instância da generalização obtida para os tipos das definições apresentadas no Exemplo 24.

Exemplo 25. Considere que a seguinte definição foi acrescentada no escopo onde todas as declarações introduzidas no Exemplo 24 estejam visíveis.

```
map::(Int → Int) → Int → Int
map f x = f x
```

O tipo desta definição de `map` não é uma instância da generalização obtida a partir das definições do Exemplo 24. Sendo assim, devemos calcular uma nova generalização e as restrições correspondentes para a classe e as instâncias. A generalização dos tipos

$$\texttt{(Map c)} \Rightarrow \texttt{(a} \rightarrow \texttt{b)} \rightarrow \texttt{c a} \rightarrow \texttt{c b}$$

e

$$\texttt{(Int} \rightarrow \texttt{Int)} \rightarrow \texttt{Int} \rightarrow \texttt{Int}$$

é o seguinte tipo:

$$\texttt{(Map a b c d)} \Rightarrow \texttt{(a} \rightarrow \texttt{b)} \rightarrow \texttt{c} \rightarrow \texttt{d}$$

e são geradas as seguintes restrições de classe e instâncias:

$$\Theta^{cls}(\texttt{Map}) = \forall\texttt{a b c d.Map a b c d}$$

e

$$\Theta^{ins}(\texttt{Map}) = \left\{ \begin{array}{c} \forall\texttt{a b.}\ \ \texttt{Map a b [a] [b]} \\ \forall\texttt{a b.}\ \ \texttt{Map a b (Tree a) (Tree b)} \end{array} \right. , \ \texttt{Map Int Int Int Int}, \left. \right\}$$

Note que as instâncias de `Map`, criadas automaticamente para listas e para árvores, são modificadas de modo a refletir a inclusão da nova definição de `map`.

5.2 Generalização Mínima

Esta seção apresenta a definição de uma função para calcular a generalização mínima (ínfimo) de um conjunto de tipos simples, de acordo com as ordens parciais definidas na Seção 4.6.

5.2.1 Semi-Reticulado de Expressões de Tipos Simples

Dizemos que $[\mu] = [\mu_1] \wedge [\mu_2]$, ou que $\mu = \mu_1 \wedge \mu_2$ (μ é o ínfimo 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 ínfimo de um conjunto finito não-vazio $X \subseteq \mathbf{T}_{\equiv}$ ($\bigwedge X$), é definido como:

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

A tupla ($\mathbf{T}_{\equiv}$,$\bigwedge$,α), onde $\mathbf{T}_{\equiv}$ é o conjunto de todos os tipos simples, $\bigwedge$ é a operação de ínfimo e α é o elemento mínimo da ordem parcial $\leq_{\mathbf{T}}$ sobre tipos simples, forma um semi-reticulado [Davey & Priestly, 1990] com as seguintes propriedades (provadas no Apêndice A):

- Para todo $\mu_1, \mu_2 \in \mathbf{T}_{\equiv}$, existe $\mu = \mu_1 \wedge \mu_2$. Isto é, todo par de elementos de $\mathbf{T}_{\equiv}$ possui um ínfimo.
- Para todo $X \subseteq \mathbf{T}_{\equiv}$, $X \neq \emptyset$, existe $\bigwedge X$, i.e. todo subconjunto de $\mathbf{T}_{\equiv}$ possui um ínfimo.
- Toda cadeia decrescente[1] de $(\mathbf{T}_{\equiv}, \leq_{\mathbf{T}})$ é finita.

A Figura 5.1 apresenta um exemplo de um fragmento de um semi-reticulado de expressões de tipo simples.

5.2.2 Cálculo do Generalização Mínima de Tipos Simples

A função para o cálculo da generalização mínima (ínfimo) de tipos simples, *lgen*, é definida recursivamente sobre a estrutura de dois tipos simples τ_1 e τ_2 fornecidos como entrada. Esta função utiliza um mapeamento finito φ de tipos simples em variáveis de tipos, para armazenar generalizações previamente realizadas. Como exemplo, considere os seguintes tipos $\tau_1 = C\ C_1\ C_1$ e $\tau_2 = C\ C_2\ C_2$ e um mapeamento finito φ. Temos

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

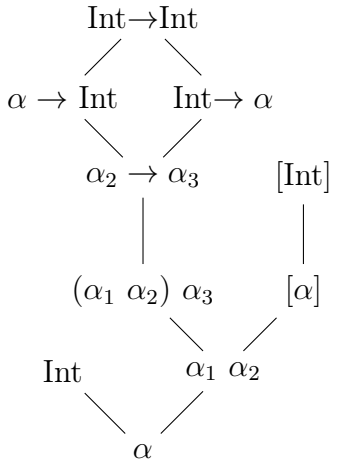

Figura 5.1. Fragmento de Semi-Reticulado de Tipos Simples

que $lgen\ \tau_1\ \tau_2 = (C\ \alpha\ \alpha, \varphi')$, onde α é uma nova variável e φ' é um novo mapeamento finito, onde o par de tipos simples (C_1, C_2) é associado com a variável α. Tal associação significa que os tipos C_1 e C_2 foram generalizados para a variável de tipos α e, sempre que estes tipos ocorrerem novamente, ao invés de gerar uma nova variável para a generalização destes, é retornada a variável α que está associada a este par de tipos no mapeamento finito φ. Sempre que um par de tipos (τ_1, τ_2) é encontrado no mapeamento φ a variável associada a eles é retornada; caso contrário, uma nova varável α é criada e o mapeamento φ é atualizado para refletir esta nova generalizadação.

A notação $\varphi\,[(\tau_1, \tau_2) \mapsto \alpha]$ denota o mapamento finito φ' que difere de φ apenas por incluir a associação $(\tau_1, \tau_2) \mapsto \alpha$ (para quaisquer outros pares de tipos (τ_a, τ_b) o resultado de φ e φ' é idêntico).

A definição da função *lgen*é apresentada na Figura 5.2. Nesta definição, a meta-variável X é utilizada para denotar uma variável ou um construtor de tipos qualquer.

A definição de uma função para calcular o ínfimo de um conjunto de tipos simples, *lcg*, é uma extensão direta da função apresentada na Figura 5.2.

$$\begin{array}{lcl} lcg\ \{\tau\} & = & \tau \\ lcg\ \{\tau\} \cup \Sigma & = & lgen\ (\tau,\ \tau')\ \emptyset \\ & & \quad \text{onde: } \tau' = lcg\ \Sigma \end{array}$$

Propriedades sobre a ordem parcial de tipos simples são enunciadas e demonstradas no Apêndice A. Além disso, uma formalização do algoritmo apresentado na figura 5.2, utilizando o assistente de provas Coq [Bertot & Castéran, 2004], é apresentada no Apêndice B.

$$
\begin{array}{lcl}
lgen\ (X_1, X_2)\ \varphi & = & \begin{cases} (X_1, \varphi) & \text{se } X_1 = X_2 \\ (\alpha, \varphi) & \text{se } X_1 \neq X_2 \text{ e } [(X_1, X_2) \mapsto \alpha] \in \varphi \\ (\alpha, \varphi') & \text{caso contrário, onde:} \\ & \quad \alpha \text{ é uma nova variável} \\ & \quad \varphi' = \varphi\,[(X_1, X_2) \mapsto \alpha] \end{cases} \\
\\
lgen\ (\tau_{11}\,\tau_{12}, \tau_{21}\,\tau_{22})\ \varphi & = & (\tau\ \tau', \varphi'),\ \text{onde:}\quad (\tau, \varphi_1) = lgen\ (\tau_{11}, \tau_{21})\ \varphi \\
& & \qquad\qquad\qquad\quad\ (\tau', \varphi') = lgen\ (\tau_{12}, \tau_{22})\ \varphi_1 \\
\\
lgen\ (\tau_1, \tau_2)\ \varphi & = & \begin{cases} (\alpha, \varphi) & \text{se } [(\tau_1, \tau_2) \mapsto \alpha] \in \varphi \\ (\alpha, \varphi') & \text{caso contrário, onde} \\ & \quad \alpha \text{ é uma nova variável} \\ & \quad \varphi' = \varphi\,[(\tau_1, \tau_2) \mapsto \alpha] \end{cases}
\end{array}
$$

Figura 5.2. Definição da Função para Cálculo do Ínfimo de Tipos Simples

5.3 Formalizando Classes de Tipos Opcionais em Haskell

Para formalizar a definição opcional de classes de tipos, é necessário estender a sintaxe da linguagem apresentada na Figura 4.1, para considerar declarações de funções. A sintaxe considerada para a declaração de funções é apresentada na Figura 5.3.

Anotações de Tipo	φ	::=	$:: \sigma$
Funções	fun	::=	$x = e\,\varphi \mid x = e$

Figura 5.3. Sintaxe de Declarações

Uma definição de função é formada por um identificador (o nome da função), uma expressão e, opcionalmente, uma anotação de tipo.

Além disso, o contexto Θ, que armazena informações sobre restrições geradas a partir de definições de classes e instâncias, também deve ser estendido, para armazenar os tipos (inferidos ou anotados) de funções sobrecarregadas sem definições de classes. Isto é necessário para a construção de restrições de instâncias, quando uma nova definição modifica a generalização mínima de um conjunto de declarações. Consideramos que o contexto Θ é formado então por 3 componentes: $\Theta = (\Theta^{cls}, \Theta^{ins}, \Theta^{o})$, onde Θ^{cls} e Θ^{ins} contêm restrições de classes e instâncias, respectivamente e o terceiro componente, Θ^{o}, é uma função finita de nomes de classes em conjuntos de tipos, que armazena os tipos das definições de nomes sobrecarregados que não fazem parte de classes. Utilizamos a notação $\Theta^{o}(C)$ para representar o conjunto de tipos associados a um nome de

classe C e $\Theta^o[C \mapsto \Sigma]$ para representar a atualização do conjunto de tipos associado a C para o conjunto de tipos Σ.

Antes de introduzirmos as funções utilizadas para atualizar o contexto Θ e gerar novas restrições de classes e instâncias, fazem-se necessárias algumas definições. A função *renamingVars* é responsável por determinar um conjunto de variáveis que não foram introduzidas no processo de generalização de tipos. Isto é feito da seguinte maneira: Seja τ a generalização de $\{\tau_i\}^{i=1..n}$. Definimos que $\alpha \in tv(\tau)$ não foi gerada durante a generalização de $\{\tau_i\}^{i=1..n}$ se existe uma variável α' e uma substituição S tal que $\alpha' \in tv(\tau_i)$ e $S\,\alpha = \alpha'$.

$$renamingVars(\tau, \{\tau_i\}^{i=1..n}) = \{\alpha \in tv(\tau) \mid S\alpha = \alpha_i, \alpha_i \in tv(\tau_i), S\tau = \tau_i\}$$

Dados dois tipos σ e σ', a função $\sigma \wedge \sigma'$ calcula a generalização destes dois tipos utilizando a função *lcg* (representada por $\bigwedge$), definida na Seção 5.2.

$$\begin{array}{lcl}(\forall\,\overline{\alpha}.P \Rightarrow \tau) \wedge (\forall\,\overline{\alpha}'.P \Rightarrow \tau') & = & \forall\,\overline{\alpha}_1.P_1 \Rightarrow \tau_1 \\ & & \text{onde:} \\ & & \quad \tau_1 = \bigwedge\{\tau, \tau'\} \\ & & \quad V = renamingVars(\tau_1, tv(\tau_1), \{\tau, \tau'\}) \\ & & \quad P_1 = (P \cup P')\,|^*_V \\ & & \quad \overline{\alpha}_1 = tv(P_1 \Rightarrow \tau_1)\end{array}$$

A notação $\Theta\,[\sigma,\, x :: \sigma']$ denota a atualização das informações relativas a classes e instâncias, em Θ, com o objetivo de incluir restrições referentes a uma nova definição $x :: \sigma'$ de um nome x que possui, como generalização mínima de suas definições, o tipo σ. Esta operação é definida na Figure 5.4 onde: $C_x = \mathit{className}\ x$, $\Theta = (\Theta^{cls}, \Theta^{ins}, \Theta^o)$, $\forall\,\overline{\alpha}\,.\,C\,\overline{\alpha} = \Theta^{cls}(C_x)$ e $\Theta^o_1 = \Theta^o\,[C_x \mapsto \Theta^o(C_x) \cup \{\sigma'\}]$.

A função *className*, quando aplicada a uma variável, retorna um nome de classe correspondente às suas definições sobrecarregadas.

As funções *genClass* e *genInst* são responsáveis pela geração de restrições de classes e instâncias respectivamente. A função *genClass* é apresentada na Figura 5.5. Ela recebe como parâmetros um identificador x e o conjunto de todos os tipos de suas definições. A partir do conjunto dos tipos das definições de x, pode-se determinar os parâmetros da classe para este símbolo, que são exatamente as variáveis de tipo que foram criadas por generalizações, no cálculo do ínfimo do conjunto de tipos de x.

A definição de *genInst* utiliza a função *match*, que retorna uma substituição S (caso esta exista), tal que $S\,\tau = \tau'$, para tipos simples τ e τ'. Esta função é definida

$$\Theta\,[\sigma,\, x :: \sigma'] = \begin{cases} ((\Theta^{cls}, \Theta_1^{ins}, \Theta_1^{o}), \sigma) & \text{se } \sigma' \leq_\Sigma \sigma, \text{ onde:} \\ & \quad \Theta_1^{ins} = \Theta^{ins} \cup \{genInst\,(x, \sigma', \sigma, \{\forall\, \overline{\alpha}\,.\, C\, \overline{\alpha}\})\} \\ ((\Theta_1^{cls}, \Theta_1^{ins}, \Theta_1^{o}), \sigma_1) & \text{caso contrário, onde:} \\ & \quad \sigma_1 = \sigma \wedge \sigma' \\ & \quad \Sigma = \Theta_1^{opt}(C_x) \\ & \quad \Theta_1^{cls} = (\Theta^{cls} - \Theta^{cls}(C_x)) \cup \{genClass\,(x, \Sigma)\} \\ & \quad \Theta_1^{ins} = (\Theta^{ins} - \Theta^{ins}(C_x)) \cup \\ & \qquad \{genInst\,(x, \sigma_i, \sigma_1, \Sigma) \mid \sigma_i \in \Sigma\} \end{cases}$$

Figura 5.4. Função para Atualização de Θ

$$\begin{array}{rcl} genClass(x, \Sigma) & = & \forall\, \overline{\alpha}.\, C_x\, \overline{\alpha} \\ & & \text{onde:} \\ & & \quad C_x = className(x) \\ & & \quad \tau = \bigwedge_{\tau_i \in \Sigma} \tau_i \\ & & \quad V_r = renamingVars(\tau, \Sigma) \\ & & \quad \overline{\alpha} = tv(\tau) - V_r \end{array}$$

Figura 5.5. Definição da Função *genClass*.

em termos da função *mgu*, que calcula o unificador mais geral de dois tipos, onde o primeiro parâmetro tem suas variáveis de tipo substituídas por novas constantes de Skolem.

$$\begin{array}{rcl} match(\tau, \tau') & = & mgu(S\,\tau, \tau') \\ & & \text{onde:} \\ & & \quad \overline{\alpha} = tv\,\tau \\ & & \quad S = [\overline{\alpha \;\mapsto\; K}] \\ & & \quad \overline{K} \text{ são novas constantes de Skolem.} \end{array}$$

A função para geração de restrições de instâncias é apresentada na Figura 5.6. Ela recebe como parâmetros um identificador x, um tipo da definição de x para a qual uma restrição de instância será retornada, a generalização mínima dos tipos de x e um conjunto dos tipos de todas as definições deste símbolo, Σ. Intuitivamente, ela calcula o conjunto de tipos que foram generalizados durante o cálculo da generalização mínima de x, σ_2. A restrição de instância é formada a partir destes tipos e das restrições de σ_1, o tipo para o qual a instância está sendo gerada.

O processo que computa classes e instâncias a partir de definições sobrecarregadas de um símbolo é apresentado na Figura 5.7, como um conjunto de julgamentos da forma

$$
\begin{array}{lcl}
genInst(x, \sigma_1, \sigma_2, \Sigma) & = & \forall\, \overline{\alpha}\,.\, P_1 \;\Rightarrow\; C\,\overline{\mu} \\
 & & \text{onde:} \\
 & & \quad \sigma_1 = \forall\, \overline{\alpha}_1\,.\, P_1 \;\Rightarrow\; \tau_1 \\
 & & \quad \sigma_2 = \forall\, \overline{\alpha}_2\,.\, P_2 \;\Rightarrow\; \tau_2 \\
 & & \quad C = className\ x \\
 & & \quad S = match\,(\tau_2, \tau_1) \\
 & & \quad V = tv\,(P_2 \;\Rightarrow\; \tau_2) - renamingVars(\tau_2, \Sigma) \\
 & & \quad S' = S\,|^*_V \\
 & & \quad \overline{\mu} = \{S'\,\alpha \,|\, S'\,\alpha \;\neq\; \alpha\} \\
 & & \quad \overline{\alpha} = tv\,\overline{\mu}
\end{array}
$$

Figura 5.6. Definição da Função *genInst*.

$\Theta \mid \Gamma \vdash^{opt} \overline{fun} \rightsquigarrow \Theta'; \Gamma'$, que utiliza o algoritmo para inferência de tipos (definido na Seção 4.11).

Uma derivação de $\Theta \mid \Gamma \vdash^{opt} \overline{fun} \rightsquigarrow \Theta'; \Gamma'$ denota que, a partir dos contextos Θ, Γ e de uma sequência de definições de funções $\overline{fun}$ são obtidos novos contextos Θ' e Γ', tais que: 1) Θ' possui as restrições de classes / instâncias induzidas pelas definições de funções $\overline{fun}$ e 2) Γ' possui o tipo de cada função definida (sobrecarregada ou não).

As regras `OPT-End` e `OPT-Many` lidam com conjuntos de definições de funções que sejam vazios ou que contenham mais de uma definição de função, respectivamente, e as demais regras consideram uma única definição de função. Duas regras (`OPT-One-1` e `OPT-One-4`) lidam com funções sem anotações de tipos e as outras duas com funções que possuem anotações de tipos. Para o caso de funções com anotações de tipo, as variáveis de tipos presentes nas anotações devem ser substituídas por constantes de Skolem.

O exemplo a seguir ilustra a utilização destas definições.

Exemplo 26. Considere novamente o Exemplo 24, onde são apresentadas definições da função `map` com os seguintes tipos $\Sigma = \{\forall\, a\, b.\, (a \to b) \to [a] \to [b], \forall\, a\, b.\, (a \to b) \to \texttt{T}\ a \to \texttt{T}\ b\}$, onde `T` abrevia o construtor de tipos `Tree`.

Seja `Map` $= className(\texttt{map})$ e $\tau' = \bigwedge \Sigma = (a \to b) \to c\,a \to c\,b$. Então, temos que a variável c é única variável introduzida pela generalização dos tipos em Σ, pois: $tv(\tau') - renamingVars(\tau', \Sigma) = \{a, b, c\} - \{a, b\} = \{c\}$. Portanto, de acordo com a definição da função *genClass*, a restrição de classe gerada para as definições de `map` do Exemplo 24 é $\forall c\,.\,\texttt{Map}\ c$. As restrições de instância são geradas a partir das substituições

$$S_{[\,]} = \{c \mapsto\ \texttt{[ ]}\}$$
$$S_{\texttt{T}} = \{c \mapsto\ \texttt{T}\}$$

$$\boxed{\Theta \mid \Gamma \vdash^{opt} \overline{fun} \rightsquigarrow \Theta'; \Gamma'}$$

$$\frac{}{\Theta \mid \Gamma \vdash^{opt} \emptyset \rightsquigarrow \Theta; \Gamma}\ \texttt{OPT-End}$$

$$\frac{\Theta \mid \Gamma \vdash^{opt} fun \rightsquigarrow \Theta_1; \Gamma_1 \quad \Theta_1 \mid \Gamma_1 \vdash^{opt} \overline{fun} \rightsquigarrow \Theta'; \Gamma'}{\Theta \mid \Gamma \vdash^{opt} fun, \overline{fun} \rightsquigarrow \Theta'; \Gamma'}\ \texttt{OPT-Many}$$

$$\frac{\begin{array}{cc} x \notin dom(\Gamma) & \Gamma' = \Gamma, x :: \sigma \\ \Theta \mid \Gamma, x :: [\overline{\alpha \mapsto K}]\,\sigma \vdash_{\mathtt{I}} e :: (\sigma', \Gamma_1) & \overline{K} \text{ são novas constantes de Skolem} \end{array}}{\Theta \mid \Gamma \vdash^{opt} x = e :: \sigma \rightsquigarrow \Theta; \Gamma'}\ \texttt{OPT-One-1}$$

$$\frac{\begin{array}{c} x \notin dom(\Gamma) \\ \Theta \mid \Gamma \vdash_{\mathtt{I}} e :: (\sigma, \Gamma_1) \qquad \Gamma' = \Gamma, x :: \sigma \end{array}}{\Theta \mid \Gamma \vdash^{opt} x = e \rightsquigarrow \Theta; \Gamma'}\ \texttt{OPT-One-2}$$

$$\frac{\begin{array}{cc} \Theta \mid \Gamma \vdash_{\mathtt{I}} e :: (\sigma', \Gamma_1) & \Gamma(x) = \sigma \\ (\Theta'; \sigma_1) = \Theta\,[\sigma,\, x :: \sigma'] & \Gamma' = \Gamma, x :: \sigma_1 \end{array}}{\Theta \mid \Gamma \vdash^{opt} x = e \rightsquigarrow \Theta'; \Gamma'}\ \texttt{OPT-One-3}$$

$$\frac{\begin{array}{cc} \Theta \mid \Gamma, x :: [\overline{\alpha \mapsto K}]\,\sigma \vdash_{\mathtt{I}} e :: (\sigma', \Gamma_1) & \overline{K} \text{ são novas constantes de Skolem} \\ \Gamma(x) = \sigma & (\Theta'; \sigma_1) = \Theta\,[\sigma,\, x :: \sigma'] \quad \Gamma' = \Gamma, x :: \sigma_1 \end{array}}{\Theta \mid \Gamma \vdash^{opt} x = e :: \sigma \rightsquigarrow \Theta'; \Gamma'}\ \texttt{OPT-One-4}$$

Figura 5.7. Geração de Classes a partir de Funções

obtidas pelo casamento (matching) de $\tau' = (a \to b) \to c\,a \to c\,b$ com os tipos de `map` para listas, $\tau_{\mathtt{[\,]}} = (d \to e) \to [d] \to [e]$, e para árvores, $\tau_{\mathtt{T}} = (d \to e) \to \mathtt{T}\ d \to \mathtt{T}\ e$, respectivamente, eliminando destas todos os pares de renomeamento de variáveis. A partir de $S_{\mathtt{[\,]}}$, podemos obter $\mu_{\mathtt{[\,]}} = \mathtt{[\ ]}$ e, com isso, construir a restrição de instância `Map [ ]`, para a definição sobrecarrega da função `map` para listas. A restrição `Map T` para árvores é obtida da mesma maneira.

Capítulo 6

Conclusão e Trabalhos Futuros

Neste trabalho apresentamos um sistema de tipos e um algoritmo de inferência para Haskell que provê suporte a classes de tipos com múltiplos parâmetros sem a necessidade de extensões. Além disso, o trabalho também proprõe a definição de símbolos sobrecarregados sem a necessidade de uma declaração de classe.

Um problema recorrente a algoritmos de inferência para Haskell é a imposição de restrições a definições de classes e instâncias para garantir terminação. Neste trabalho um algoritmo que não impõe restrições sintáticas sobre definições de classes e instâncias foi elaborado e a terminação deste foi provada.

No Capítulo 4 foram apresentados dois sistemas de tipo: um que permite instanciações independentes de contexto e um que permite apenas instanciações dependentes de contexto. Como ressaltado nesse mesmo capítulo, o algoritmo de inferência de tipos é correto mas não completo em relação ao primeiro sistema de tipos. Esse fato é decorrente da utilização de uma versão não restrita do problema de satisfazibilidade de restrições e da possibilidade de instanciações independentes de contexto que podem ser utilizadas para obter derivações de tipo diferentes para uma mesma expressão, que possuem traduções diferentes. Por sua vez, o sistema de tipos que utiliza instanciações dependentes de contexto e a versão restrita do problema de satisfazibilidade é correto e completo em relação ao algoritmo de inferência proposto.

Um protótipo do algoritmo de inferência de tipos proposto foi implementado e está disponível no seguinte repositório:

http://github.com/rodrigogribeiro/mptc

tal protótipo foi submetido a diversos testes envolvendo bibliotecas que utilizam classes com vários parâmetros e dependências funcionais; além de diversos exemplos presentes

na literatura, onde comportou-se conforme o esperado. Até o presente momento, somente instâncias de problemas de satisfazibilidade construídas a partir da codificação de Problemas de Correspondência de Post solúveis fazem com que o algoritmo não apresente uma resposta correta.

Demonstrações de propriedades de correção do algoritmo de inferência proposto foram realizadas. Além disso, foram demonstradas diversas propriedades de terminação e corretude dos algoritmos utilizados para o cálculo da generalização mínima de um conjunto de tipos simples. Tais algoritmos são utilizadas para prover suporte a definição de símbolos sobrecarregados sem a declaração de classes e instâncias.

Possíveis trabalhos futuros envolvem:

- A formalização dos algoritmos propostos em um assistente de provas. Apesar desse trabalho possuir a formalização dos algoritmos propostos, a mecanização de tais provas pode permitir a extração de algoritmos corretos por construção.

- Prover suporte a novos recursos da linguagem Haskell: tipos de dados algébricos generalizados e famílias de tipos. Apesar de nesse trabalho argumentarmos contra o uso de famílias de tipos para permitir a utilização de classes de tipos com vários parâmetros, tal extensão possibilita uma forma restrita de tipos dependentes em Haskell.

- Integração do *front-end* desenvolvido a um back-end para produzir um compilador Haskell que implemente o sistema de tipos descrito nesta tese.

Apêndice A

Provas

Neste apêndice são enunciados e provados diversos teoremas sobre o trabalho desenvolvido nesta tese.

A.1 Ordens Parciais

A.1.1 Pré-Ordem de Tipos Simples

Lema 1. *A relação $\preceq_{\boldsymbol{T}}$ é reflexiva.*

Demonstração. Para mostrar que $\mu \preceq_{\mathbf{T}} \mu$ basta considerar $S = id$. □

Lema 2. *A relação $\preceq_{\boldsymbol{T}}$ é transitiva.*

Demonstração. Suponha μ_1, μ_2 e μ_3 arbitrários tais que $\mu_1 \preceq_{\mathbf{T}} \mu_2$ e $\mu_2 \preceq_{\mathbf{T}} \mu_3$. Pela definição de $\preceq_{\mathbf{T}}$ temos que devem existir S_1 e S_2 tais que $\mu_1 = S_1\,\mu_2$ e $\mu_2 = S_2\,\mu_3$. Sendo assim, temos que $\mu_1 = S\,\mu_3$, onde $S = S_2 \circ S_1$. □

Corolário 1 (Pré-Ordem de Tipos Simples)**.** *A relação $\preceq_{\boldsymbol{T}}$ é uma pré-ordem.*

Demonstração. Corolário dos Lemas 1 e 2. □

A.1.2 Equivalência Módulo Renomeação de Variáveis

Lema 3. *A relação $\equiv_{\boldsymbol{T}}$ é reflexiva.*

Demonstração. Consequência direta do Lema 1 e da definição de $\equiv_{\mathbf{T}}$. □

Lema 4. *A relação $\equiv_{\boldsymbol{T}}$ é transitiva.*

Demonstração. Consequência direta do Lema 2 e da definição de $\equiv_{\mathbf{T}}$. □

Lema 5. *A relação $\equiv_{\boldsymbol{T}}$ é simétrica.*

Demonstração. Consequência direta da definição de $\equiv_{\mathbf{T}}$. □

Corolário 2. *A relação $\equiv_{\boldsymbol{T}}$ é uma relação de equivalência.*

Demonstração. Corolário dos Lemas 3, 4 e 5. □

A.1.3 Ordem Parcial de Tipos Simples

Lema 6. *A relação $\leq_{\boldsymbol{T}}$ é antisimétrica.*

Demonstração. Suponha μ, μ' arbitrários tais que $\mu \leq_{\mathbf{T}} \mu'$ e $\mu' \leq_{\mathbf{T}} \mu$. A prova procederá por indução sobre a estrutura de μ e análise de casos sobre μ'. Evidentemente, se μ é uma aplicação e μ' um construtor de tipos (ou vice-versa), ou μ e μ' são diferentes construtores de tipos temos que as hipóteses $\mu \leq_{\mathbf{T}} \mu'$ e $\mu' \leq_{\mathbf{T}} \mu$ são falsas e o resultado desejado é trivialmente válido. Considere, então, os seguintes casos:

- Caso $\mu = \alpha$: Se $\mu' = \alpha'$, temos que o resultado é imediato, uma vez que $\alpha \equiv_{\mathbf{T}} \alpha'$. Caso μ' seja uma aplicação ou um construtor de tipos, temos que a hipótese $\mu' \leq_{\mathbf{T}} \mu$ é falsa e o resultado é trivialmente válido.
- Caso $\mu = T$: Se $\mu' = \alpha'$ ou $\mu' = \mu_{11}\,\mu_{12}$, a hipótese $\mu \leq_{\mathbf{T}} \mu'$ é falsa e o resultado trivialmente válido. Caso $\mu' = T$, temos que $\mu \equiv_{\mathbf{T}} \mu'$.
- Caso $\mu = \mu_{11}\,\mu_{12}$: Evidentemente, se μ' é uma variável ou aplicação, o resultado é imediato, pois a hipótese $\mu \leq_{\mathbf{T}} \mu'$ é falsa. Logo, considere que $\mu' = \mu_{21}\,\mu_{22}$. Pela hipótese de indução, temos que $\mu_{11} \equiv_{\mathbf{T}} \mu_{21}$ e $\mu_{21} \equiv_{\mathbf{T}} \mu_{22}$ e, portanto, $\mu \equiv_{\mathbf{T}} \mu'$, conforme requerido.

□

Corolário 3 (Ordem parcial de tipo simples)**.** *A relação $\leq_{\boldsymbol{T}}$ é uma ordem parcial.*

Demonstração. Corolário dos Lemas 3, 4 e 6. □

Lema 7 (Elemento mínimo da ordem parcial sobre tipos simples)**.** *Seja μ um tipo simples qualquer. Então $\alpha \leq_{\boldsymbol{T}} \mu$, onde α é uma variável de tipo.*

Demonstração. Suponha μ e α arbitrários. Para mostrar que $\alpha \leq_{\mathbf{T}} \mu$ basta existir uma substituição S tal que $S\,\alpha = \mu$. Basta considerar $S = [\alpha \mapsto \mu]$. □

A.1.4 Semi-Reticulado de Expressões de Tipo Simples

Teorema 1 (Existência de ínfimo para pares de tipos simples)**.** *Para todo* $\mu_1, \mu_2 \in \boldsymbol{T}_{\equiv}$*, existe* $\mu = \mu_1 \wedge \mu_2$*.*

Demonstração. Por indução sobre a estrutura de μ_1 e análise de casos sobre a estrutura de μ_2. Nos casos em que a estrutura de μ_1 é diferente da estrutura de μ_2 o resultado é imediato, já que o único ínfimo possível é alguma variável de tipo α. Considere, então, os seguintes casos:

- Caso μ_1 e μ_2 são variáveis de tipos. Então, $\mu_1 = \alpha_1$ e $\mu_2 = \alpha_2$ para variáveis α_1 e α_2. Seja $\mu = \alpha$. Como $S_1 = [\alpha \mapsto \alpha_1]$ e $S_2 = [\alpha \mapsto \alpha_2]$, temos que $\alpha_1 \leq_{\mathbf{T}} \alpha$ e $\alpha_2 \leq_{\mathbf{T}} \alpha$. Suponha μ' arbitrário tal que $\alpha_1 \leq_{\mathbf{T}} \mu'$ e $\alpha_2 \leq_{\mathbf{T}} \mu'$. Neste caso, devem existir S_1' e S_2' tais que $S_1' \mu' = \alpha_1$ e $S_2' \mu' = \alpha_2$. Pela definição de aplicação de substituições, temos que $\mu' = \alpha'$, para alguma variável α'. Uma vez que $\alpha \equiv_{\mathbf{T}} \alpha'$, temos que α é o ínfimo de α_1 e α_2.

- Caso μ_1 e μ_2 são construtores de tipos. Então $\mu_1 = T_1$ e $\mu_2 = T_2$ para construtores de tipos T_1 e T_2. Considere os seguintes casos:

  - Caso $T_1 = T_2$: Seja $\mu = T_1$. É evidente que se $T_1 = T_2$, $\mu = T_1$ é tal que $\mu \leq_{\mathbf{T}} T_1$, $\mu \leq_{\mathbf{T}} T_2$ e é o menor μ com esta propriedade.

  - Caso $T_1 \neq T_2$: Seja $\mu = \alpha$, para alguma variável de tipo α. Uma vez que $S_1 = [\alpha \mapsto T_1]$ e $S_2 = [\alpha \mapsto T_2]$, temos que $T_1 \leq_{\mathbf{T}} \alpha$ e $T_2 \leq_{\mathbf{T}} \alpha$. Suponha μ' arbitrário tal que $T_1 \leq_{\mathbf{T}} \mu'$ e $T_2 \leq_{\mathbf{T}} \mu'$. Neste caso, devem existir S_1' e S_2' tais que $S_1' \mu' = T_1$ e $S_2' \mu' = T_2$. Pela definição de aplicação de substituições, temos que μ' não deve ser uma aplicação ou um construtor de tipos, pois isso contradiria a suposição de que $T_1 \neq T_2$. Portanto $\mu' = \alpha'$, para alguma variável α'. Como $\alpha \equiv_{\mathbf{T}} \alpha'$, temos que $\alpha = T_1 \wedge T_2$ conforme requerido.

- Caso μ_1 e μ_2 são aplicações. Então $\mu_1 = (\mu_{11}\ \mu_{12})$ e $\mu_2 = (\mu_{21}\ \mu_{22})$ para tipos μ_{11}, μ_{12}, μ_{21} e μ_{22}. Pela hipótese de indução, devem existir μ_l e μ_r tais que $\mu_l = \mu_{11} \wedge \mu_{21}$ e $\mu_r = \mu_{12} \wedge \mu_{22}$. Seja $\mu = \mu_l\ \mu_r$. Uma vez que $\mu_l = \mu_{11} \wedge \mu_{21}$ e $\mu_r = \mu_{12} \wedge \mu_{22}$, temos que existem S_1 e S_2 tais que $S_1\, \mu_l = \mu_{11}$ e $S_2\, \mu_r = \mu_{12}$. Com isso, temos que $(S_2 \circ S_1)(\mu_l\ \mu_r) = \mu_1$. De maneira similar, temos que existem S_1' e S_2' tais que $S_1'\, \mu_l = \mu_{21}$, $S_2'\, \mu_r = \mu_{22}$ e $(S_2' \circ S_1')(\mu_l\ \mu_r) = \mu_2$. Portanto, temos que $\mu_1 \leq_{\mathbf{T}} \mu_l\ \mu_r$ e $\mu_2 \leq_{\mathbf{T}} \mu_l\ \mu_r$. Suponha μ' arbitrário tal que $\mu_1 \leq_{\mathbf{T}} \mu'$ e $\mu_2 \leq_{\mathbf{T}} \mu'$. Uma vez que μ_1 e μ_2 são aplicações, μ' também o é; e portanto existem μ_a e μ_b tais que $\mu' = \mu_a\ \mu_b$. Como $\mu_1 = \mu_{11}\ \mu_{12}$ e $\mu_1 \leq_{\mathbf{T}} \mu'$ temos que $\mu_{11} \leq_{\mathbf{T}} \mu_a$

e $\mu_{12} \leq_{\mathbf{T}} \mu_b$. De maneira similar para μ_2 temos que $\mu_{21} \leq_{\mathbf{T}} \mu_a$ e $\mu_{22} \leq_{\mathbf{T}} \mu_b$. Portanto, $\mu' \leq_{\mathbf{T}} \mu$, conforme requerido.

□

Teorema 2 (Existência de ínfimo para conjuntos não vazios de tipos simples)**.** *Para todo $X \subseteq \boldsymbol{T}_{\equiv}$, $X \neq \emptyset$, existe $\bigwedge X$.*

Demonstração. Indução sobre $|X|$ utilizando o Teorema 1. □

Seja $\chi(\mu)$ uma função que retorna o número de construtores de tipos e aplicações presentes em μ. Mais formalmente, $\chi : T_{\equiv} \to \mathbb{N}$:

$$\begin{aligned} \chi(\alpha) &= 0 \\ \chi(T) &= 1 \\ \chi(\mu_1\ \mu_2) &= 1 + \chi(\mu_1) + \chi(\mu_2) \end{aligned}$$

Lema 8. *Se $\mu_1 \leq_{\boldsymbol{T}} \mu_2$ e $\mu_1 \notin [\mu_2]$ então $\chi(\mu_1) < \chi(\mu_2)$.*

Demonstração. Suponha que $\mu_1 \leq_{\mathbf{T}} \mu_2$ e $\mu_1 \notin [\mu_2]$. A prova será por indução sobre a estrutura de μ_1 e análise de casos sobre a estrutura de μ_2. Os casos nos quais $\mu_1 \equiv_{\mathbf{T}} \mu_2$ são imediatos, já que contradizem a suposição de que $\mu_1 \notin [\mu_2]$. Considere agora os seguintes casos:

1. Caso $\mu_1 = \alpha$. Neste caso, temos que $\chi(\alpha) = 0$. Evidentemente, como $\mu_2 \notin [\mu_1 = \alpha]$, temos que μ_2 é uma aplicação ou um construtor de tipos. Portanto, $\mu_1 \leq_{\mathbf{T}} \mu_2$.

2. Caso $\mu_1 = T_1$. Neste caso, temos que $\chi(T_1) = 1$. Como $\mu_1 \leq_{\mathbf{T}} \mu_2$, temos que $\mu_2 = T_1$, o que contradiz a suposição de que $\mu_1 \notin [\mu_2]$. Logo, $\chi(\mu_1) < \chi(\mu_2)$.

3. Caso $\mu_1 = \mu_{11}\ \mu_{12}$. Neste caso, temos que $\chi(\mu_{11}\ \mu_{12}) = 1 + \chi(\mu_{11}) + \chi(\mu_{12})$. Evidentemente, como $\mu_2 \notin [\mu_1 = \mu_{11}\ \mu_{12}]$, temos que μ_2 é uma variável de tipo ou uma aplicação tal que $\mu_2 = \mu_{21}\ \mu_{22}$ onde $\mu_{11} \notin [\mu_{21}]$, $\mu_{12} \notin [\mu_{22}]$, $\mu_{11} \leq_{\mathbf{T}} \mu_{21}$ e $\mu_{12} \leq_{\mathbf{T}} \mu_{22}$. Pela hipótese de indução, temos que $\chi(\mu_{11}) < \chi(\mu_{21})$ e $\chi(\mu_{12}) < \chi(\mu_{22})$. Logo, $1+\chi(\mu_{11})+\chi(\mu_{12}) < 1+\chi(\mu_{21})+\chi(\mu_{22})$ e portanto, $\chi(\mu_1) < \chi(\mu_2)$ conforme requerido.

□

Teorema 3. *Toda cadeia decrescente de $(\boldsymbol{T}_{\equiv}, \leq_{\boldsymbol{T}})$ é finita.*

Demonstração. Suponha, por contradição, que exista uma cadeia decrescente infinita de $(\mathbf{T}_{\equiv}, \leq_{\mathbf{T}})$ e esta seja $C = \mu_1 \leq_{\mathbf{T}} \mu_2 \leq_{\mathbf{T}} \ldots$. Pelo Lema 8, temos que C só pode ser infinita se a função $\chi(\mu)$ decrescer infinitamente. Mas como a imagem de χ é o conjunto $\mathbb{N}$ e esta função é definida para todo $\mu \in \mathbf{T}_{\equiv}$, temos que $(\mathbf{T}_{\equiv}, \leq_{\mathbf{T}})$ não possui cadeias decrescentes infinitas. □

Lema 9. *Sejam μ_1 e μ_2 dois tipos simples tais que $\mu = \mu_1 \wedge \mu_2$ e que $\mu' = lgen\,\mu_1\,\mu_2\,\varphi$, para algum mapeamento finito φ. Então, $\mu \equiv_{\boldsymbol{T}} \mu'$.*

Demonstração. Indução sobre a estrutura de μ_1 e análise de casos sobre a estrutura de μ_2. □

Teorema 4. *Suponha X um conjunto não vazio de tipo simples e que $\mu = \bigwedge X$ e que $\mu' = \mathrm{lcg}\ X$. Então $\mu \equiv_{\boldsymbol{T}} \mu'$.*

Demonstração. Indução sobre $|X|$ utilizando o Lema 9. □

A.2 Satisfazibilidade de Restrições

Teorema 5 (Corretude de $\vdash^{\mathtt{sats}}$)**.** *Se $\Theta \vdash^{sats} P \rightsquigarrow \mathbb{S}$ então $\Theta \Vdash S\,P$, para cada $S \in \mathbb{S}$.*

Demonstração. Por indução sobre a derivação de $\Theta \vdash^{\mathtt{sats}} P \rightsquigarrow \mathbb{S}$. O único caso interessante é o da regra `SInst`. Seja $\pi = C\,\overline{\mu}$ e $\Delta = sats(\pi, \Theta)$. Se $\Delta = \emptyset$, o teorema é trivialmente verdadeiro. Assim, suponha que $\Delta \neq \emptyset$ e que $(S, Q, C\,\overline{\mu}_0) \in \Delta$. Pela definição de *sats*, isso significa que $\forall\,\overline{\alpha}.\,P_0 \Rightarrow C\,\overline{\mu}_0 \in \Theta$, onde $\overline{\alpha} = tv(P_0 \Rightarrow C\,\overline{\mu}_0)$, e $P' \Rightarrow C\,\overline{\mu} = [\,\overline{\alpha} \mapsto \overline{\beta}\,]P_0 \Rightarrow C\,\overline{\mu}_0$. Pela regra `Inst` temos que $\Theta, P_0 \Vdash C\,\overline{\mu}_0$ é provável. Além disso, temos que $\Theta \vdash^{\mathtt{sats}} Q \rightsquigarrow \mathbb{S}_0$, onde $Q = S\,[\,\overline{\alpha} \mapsto \overline{\beta}\,]\,P_0$, e assim, pela hipótese de indução, temos que (1) $\Theta \Vdash S'\,Q$ é válido para todo $S' \in \mathbb{S}_0$. Uma vez que, $\Theta, P_0 \Vdash C\,\overline{\mu}_0$ é provável, temos pela regra `Subst`, que (2) $\Theta, S_0\,P_0 \Vdash S_0\,C\,\overline{\mu}_0$, onde $S_0 = S'\,S\,[\,\overline{\alpha} \mapsto \overline{\beta}\,]$. De (1) e (2) temos, pela regra `Trans`, que $\Theta \Vdash S_0\,C\,\overline{\mu}_0$ é' provável. Já que $S\,\overline{\mu} = S\,[\,\overline{\alpha} \mapsto \overline{\beta}\,]\,\overline{\mu}_0$, temos que $\Theta \Vdash S'\,S\,C\,\overline{\mu}$ é provável. □

Teorema 6 (Completude of $\vdash^{\mathtt{sats}}$)**.** *Se $\Theta \Vdash S\,P$ então existe $S' \in \mathbb{S}$ tal que $S'\,P = S\,P$, onde $\Theta \vdash^{sats} P \rightsquigarrow \mathbb{S}$.*

Demonstração. Indução sobre $S\,P$ em $\Theta \Vdash S\,P$. □

Teorema 7 (Corretude de $\vdash^{\mathtt{tsat}}$)**.** *Se $\Theta, \Phi_0 \vdash^{tsat} P \rightsquigarrow \mathbb{S}$ então $\Theta \vdash^{sats} P \rightsquigarrow \mathbb{S}$.*

Demonstração. Indução sobre a derivação de $\Theta, \Phi_0 \vdash^{\mathtt{tsat}} P \rightsquigarrow \mathbb{S}$. □

Teorema 8 (Terminação de $\vdash^{\texttt{tsat}}$)**.** *A computação de* $\Theta, \Phi_0 \vdash^{tsat} P \rightsquigarrow \mathbb{S}$ *termina para qualquer* P *e* Θ*.*

Demonstração. Para provar que a computação de $\Theta, \Phi_0 \vdash^{\texttt{tsat}} P \rightsquigarrow \mathbb{S}$ termina para qualquer P, considere que uma recursão infinita somente pode acontecer se um número infinito de restrições unificarem com uma cabeça de instância π_0 presente em Θ (esta é a única possibilidade, já que existem finitas instâncias em Θ). Mas isto não é possível, uma vez que para cada nova restrição π que unifica com π_0, temos que, pela definição de $\Phi[\pi_0, \pi]$, que $\Phi(\pi_0)$ é atualizado para um valor distindo de todos os valores anteriores (caso contrário $\Phi[\pi_0, \pi]$ produziria *Fail* e a computação termina). A terminação de $\Theta, \Phi_0 \vdash^{\texttt{tsat}} P \rightsquigarrow \mathbb{S}$ segue do fato que $\Phi(\pi_0)$ pode assumir apenas um conjunto finito de valores, para todo π_0. Isto pode constatado pelo fato de que, para qualquer $\Phi(\pi_0) = (I, \Pi)$, a inserção de um nova restrição em Π diminui $k - k'$, onde k é o número finito de todos os possíveis valores que pode ser inseridos em Π e k' é a cardinalidade de Π. O decréscimo deste valor faz com que Φ também diminua (já que existem somente finitas cabeças de instância π_0 em Θ). De maneira similar, a cada passo deve existir algum i tal que $I.v_i$ diminui, e isso só pode acontecer um número finito de vezes. Portanto, podemos concluir que a computação de $\Theta, \Phi_0 \vdash^{\texttt{tsat}} P \rightsquigarrow \mathbb{S}$ sempre termina para qualquer P. □

A.3 Redução de Contexto

Teorema 9 (Corretude de $\vdash^{\texttt{simp}}$)**.** *Se* Θ, $\Phi \vdash^{simp} P \rightsquigarrow Q$ *é provável, então* $\Theta, Q \Vdash P$ *é provável e* Q *não pode ser mais simplificado, i.e.* Θ, $\Phi \vdash^{simp} Q \rightsquigarrow Q$*.*

Demonstração. Indução sobre a derivação de Θ, $\Phi \vdash^{\texttt{simp}} P \rightsquigarrow Q$. □

A.4 Inferência de Tipos

Teorema 10 (Corretude de $\vdash_{\texttt{I}}$)**.** *Se* $\Theta \mid \Gamma \vdash_I e : (P \Rightarrow \tau, S)$ *então* $\Theta \mid \Gamma \vdash e : P \Rightarrow \tau$*.*

Demonstração. Por indução sobre a derivação de $\Theta \mid \Gamma \vdash_{\texttt{I}} e : (P \Rightarrow \tau, S)$, realizando análise de casos sobre a última regra utilizada nesta derivação. Considere os seguintes casos:

- Caso $(\texttt{VAR}_{\texttt{I}})$: Neste caso, temos que $e = x$, para alguma variável x e que existe $\sigma = \forall \overline{\alpha}.Q \Rightarrow \tau'$ tal que $\Gamma(x) = \sigma$ e que $P \Rightarrow \tau = [\overline{\beta} \mapsto \overline{\alpha}]\, Q \Rightarrow \tau$. Logo,

temos que $\sigma \leq_\Sigma P \Rightarrow \tau$ e, portanto, pela regra (VAR) podemos deduzir que $\Theta \mid \Gamma \vdash x : P \Rightarrow \tau$, conforme requerido.

- Caso ($\mathtt{ABS_I}$): Neste caso, temos que $e = \lambda x.e'$, $\tau = S\,\alpha$ e que $\Theta \mid \Gamma\,[x : \alpha] \vdash_\mathtt{I} e' : (P \Rightarrow \tau_1, S)$. Pela hipótese de indução, temos que $\Theta \mid \Gamma\,[x : \alpha] \vdash e' : P \Rightarrow \tau_1$ e, portanto, pela regra (ABS) podemos concluir que $\Theta \mid \Gamma \vdash \lambda x.e' : P \Rightarrow \tau \to \tau_1$, conforme requerido.

- Caso ($\mathtt{APP_I}$): Neste caso, temos que $e = e_1\,e_2$, $\Theta \mid \Gamma \vdash_\mathtt{I} e_1 : (P_1 \Rightarrow \tau_1, S_1)$, $\Theta \mid \Gamma \vdash_\mathtt{I} e_2 : (P_2 \Rightarrow \tau_2, S_2)$, $\tau = S\,\alpha$, $S = S' \circ S_2 \circ S_1$ e $S = mgu(\{\tau_1 = \tau_2 \to \alpha\})$. Pela hipótese de indução, temos que $\Theta \mid \Gamma \vdash e_1 : P_1 \Rightarrow \tau_2 \to \tau$ e $\Theta \mid \Gamma \vdash e_2 : P_2 \Rightarrow \tau_2$, portanto, pela regra (APP) podemos concluir que $\Theta \mid \Gamma \vdash e_1\,e_2 : (P_1 \cup P_2) \Rightarrow \tau$, conforme requerido.

- Caso ($\mathtt{LET_I}$): Neste caso, temos que $e = \mathtt{let}\,x = e_1\,\mathtt{in}\,e_2$, $\Theta \mid \Gamma \vdash_\mathtt{I} e_1 : (Q \Rightarrow \tau', S')$, $\sigma = gen(\Theta, \Gamma, Q \Rightarrow \tau')$ e $\Theta \mid \Gamma\,[x : \sigma] \vdash_\mathtt{I} e_2 : (P \Rightarrow \tau, S)$. Pela hipótese de indução, temos que $\Theta \mid \Gamma \vdash e_1 : Q \Rightarrow \tau'$, $\sigma = gen(\Theta, \Gamma, Q \Rightarrow \tau')$, $\Theta \mid \Gamma\,[x : \sigma] \vdash e_2 : P \Rightarrow \tau$ e, portanto, pela regra (LET) temos que $\Theta \mid \Gamma \vdash \mathtt{let}\,x = e_1\,\mathtt{in}\,e_2 : P \Rightarrow \tau$, conforme requerido.

□

Teorema 11 (Corretude de $\vdash_\mathtt{I}$ em relação a $\vdash^D$)**.** *Se* $\Theta \mid \Gamma \vdash_I e : (P \Rightarrow \tau, S)$ *então existe* $\Theta \mid \Gamma \vdash^D e : (P \Rightarrow \tau, S')$ *tal que* $S \leq_\mathbb{S} S'$.

Demonstração. Por indução sobre a derivação de $\Theta \mid \Gamma \vdash_\mathtt{I} e : (P \Rightarrow \tau, S)$, realizando análise de casos sobre a última regra utilizada nesta derivação. Considere os seguintes casos:

- Caso ($\mathtt{VAR_I}$): Neste caso, temos que $e = x$, para alguma variável x e que existe $\sigma = \forall\,\overline{\alpha}.Q \Rightarrow \tau'$ tal que $\Gamma(x) = \sigma$, $P \Rightarrow \tau = [\overline{\beta} \mapsto \overline{\alpha}]\,Q \Rightarrow \tau$ e $S = id$. Logo, temos que $\sigma \leq_\Sigma P \Rightarrow \tau$ e, portanto, pela regra (VAR) podemos deduzir que $\Theta \mid \Gamma \vdash^D x : (P \Rightarrow \tau, id)$ e $id \leq_\mathbb{S} id$, conforme requerido.

- Caso ($\mathtt{ABS_I}$): Neste caso, temos que $e = \lambda x.e'$, $\tau = S\,\alpha$ e que $\Theta \mid \Gamma\,[x : \alpha] \vdash_\mathtt{I} e' : (P \Rightarrow \tau_1, S)$. Pela hipótese de indução, temos que $\Theta \mid \Gamma\,[x : \alpha] \vdash^D e' : P \Rightarrow \tau_1$ e, portanto, pela regra (ABS) podemos concluir que $\Theta \mid \Gamma \vdash^D \lambda x.e' : (P \Rightarrow \tau \to \tau_1, S)$ e $S \leq_\mathbb{S} S$, conforme requerido.

- Caso ($\mathtt{APP_I}$): Neste caso, temos que $e = e_1\,e_2$, $\Theta \mid \Gamma \vdash_\mathtt{I} e_1 : (P_1 \Rightarrow \tau_1, S_1)$, $\Theta \mid \Gamma \vdash_\mathtt{I} e_2 : (P_2 \Rightarrow \tau_2, S_2)$, $\tau = S\,\alpha$, $S = S' \circ S_2 \circ S_1$ e $S = mgu(\{\tau_1 =$

$\tau_2 \to \alpha\}$). Pela hipótese de indução, temos que $\Theta \mid \Gamma \vdash^D e_1 : P_1 \Rightarrow \tau_2 \to \tau$ e $\Theta \mid \Gamma \vdash^D e_2 : P_2 \Rightarrow \tau_2$, portanto, pela regra (APP) podemos concluir que $\Theta \mid \Gamma \vdash^D e_1\, e_2 : ((P_1 \cup P_2) \Rightarrow \tau, S \circ S' \circ S_2 \circ S_1)$ e $S \leq_{\mathbb{S}} S$, conforme requerido.

- Caso ($\texttt{LET}_\texttt{I}$): Neste caso, temos que $e = \texttt{let}\, x = e_1\, \texttt{in}\, e_2$, $\Theta \mid \Gamma \vdash_\texttt{I} e_1 : (Q \Rightarrow \tau', S')$, $\sigma = gen(\Theta, \Gamma, Q \Rightarrow \tau')$ e $\Theta \mid \Gamma\,[x : \sigma] \vdash_\texttt{I} e_2 : (P \Rightarrow \tau, S)$. Pela hipótese de indução, temos que $\Theta \mid \Gamma \vdash^D e_1 : Q \Rightarrow \tau'$, $\sigma = gen(\Theta, \Gamma, Q \Rightarrow \tau')$, $\Theta \mid \Gamma\,[x : \sigma] \vdash^D e_2 : P \Rightarrow \tau$ e, portanto, pela regra (LET) temos que $\Theta \mid \Gamma \vdash^D \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : (P \Rightarrow \tau, S)$ e $S \leq_{\mathbb{S}} S$, conforme requerido.

□

Teorema 12 (Completude de $\vdash_\texttt{I}$ em relação a $\vdash^D$)**.** *Se* $\Theta \mid \Gamma \vdash^D e : (P \Rightarrow \tau, S)$ *então existe* S' *tal que* $\Theta \mid \Gamma \vdash_I e : (P \Rightarrow \tau, S')$ *e* $S' \leq_{\mathbb{S}} S$.

Demonstração. Por indução sobre a derivação de $\Theta \mid \Gamma \vdash_\texttt{D} e : (P \Rightarrow \tau, S)$, realizando análise de casos sobre a última regra utilizada nesta derivação. Considere os seguintes casos:

- Caso (VAR): Neste caso, temos que $e = x$, para alguma variável x e que existe $\sigma = \forall\, \overline{\alpha}.Q \Rightarrow \tau'$ tal que $\Gamma(x) = \sigma$, $P \Rightarrow \tau = [\overline{\beta} \mapsto \overline{\alpha}]\, Q \Rightarrow \tau$ e $S = id$. Logo, temos que $\sigma \leq_\Sigma P \Rightarrow \tau$ e, portanto, pela regra ($\texttt{VAR}_\texttt{I}$) podemos deduzir que $\Theta \mid \Gamma \vdash_I x : (P \Rightarrow \tau, id)$, conforme requerido.

- Caso (ABS): Neste caso, temos que $e = \lambda x.e'$, $\tau = S\,\alpha$ e que $\Theta \mid \Gamma\,[x : \alpha] \vdash^\texttt{D} e' : (P \Rightarrow \tau_1, S)$. Pela hipótese de indução, temos que $\Theta \mid \Gamma\,[x : \alpha] \vdash_I e' : P \Rightarrow \tau_1$ e, portanto, pela regra ($\texttt{ABS}_\texttt{I}$) podemos concluir que $\Theta \mid \Gamma \vdash_I \lambda x.e' : (P \Rightarrow \tau \to \tau_1, S)$, conforme requerido.

- Caso (APP): Neste caso, temos que $e = e_1\, e_2$, $\Theta \mid \Gamma \vdash^\texttt{D} e_1 : (P_1 \Rightarrow \tau_1, S_1)$, $\Theta \mid \Gamma \vdash^\texttt{D} e_2 : (P_2 \Rightarrow \tau_2, S_2)$, $\tau = S\,\alpha$, $S = S' \circ S_2 \circ S_1$ e $S = mgu(\{\tau_1 = \tau_2 \to \alpha\})$. Pela hipótese de indução, temos que $\Theta \mid \Gamma \vdash_I e_1 : P_1 \Rightarrow \tau_2 \to \tau$ e $\Theta \mid \Gamma \vdash_I e_2 : P_2 \Rightarrow \tau_2$, portanto, pela regra ($\texttt{APP}_\texttt{I}$) podemos concluir que $\Theta \mid \Gamma \vdash_I e_1\, e_2 : ((P_1 \cup P_2) \Rightarrow \tau, S \circ S' \circ S_2 \circ S_1)$, conforme requerido.

- Caso ($\texttt{LET}^\texttt{D}$): Neste caso, temos que $e = \texttt{let}\, x = e_1\, \texttt{in}\, e_2$, $\Theta \mid \Gamma \vdash^\texttt{D} e_1 : (Q \Rightarrow \tau', S')$, $\sigma = gen(\Theta, \Gamma, Q \Rightarrow \tau')$ e $\Theta \mid \Gamma\,[x : \sigma] \vdash^\texttt{D} e_2 : (P \Rightarrow \tau, S)$. Pela hipótese de indução, temos que $\Theta \mid \Gamma \vdash_I e_1 : Q \Rightarrow \tau'$, $\sigma = gen(\Theta, \Gamma, Q \Rightarrow \tau')$, $\Theta \mid \Gamma\,[x : \sigma] \vdash_I e_2 : P \Rightarrow \tau$ e, portanto, pela regra ($\texttt{LET}_\texttt{I}$) temos que $\Theta \mid \Gamma \vdash_I \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : (P \Rightarrow \tau, S)$, conforme requerido.

□

A.5 Instanciações Dependentes de Contexto

Teorema 13. *Se* $\Theta \mid \Gamma \vdash^D e : (P \Rightarrow \tau, S)$ *então em todos os contextos* $\mathcal{C}[e]$ *e todo* Γ' *tal que* $\Gamma \leq_\omega \Gamma'$ *e* $\Theta \mid \Gamma' \vdash^D \mathcal{C}[e] : (P' \Rightarrow \tau', S')$ *é provável para algum* $P' \Rightarrow \tau'$ *e* S' *temos que* $S \leq_{\mathbb{S}} S'$.

Demonstração. Indução sobre $\Theta \mid \Gamma \vdash^D e : (P \Rightarrow \tau, S)$. □

A.6 Semântica

Teorema 14 (Coerência)**.** *Para quaisquer derivações* Δ, Δ' *de* $\Theta \mid \Gamma \vdash^D e : (P \Rightarrow \tau, S)$ *e* $\Theta \mid \Gamma' \vdash^D e : (P \Rightarrow \tau, S)$*, respectivamente, onde* $\Gamma(x) = \Gamma'(x)$ *para todo* x *livre in* e*, temos que* $[\![\Theta \mid \Gamma \vdash^D e : (P \Rightarrow \tau, S)]\!]_\eta = [\![\Theta \mid \Gamma' \vdash^D e : (P \Rightarrow \tau, S)]\!]_\eta$.

Demonstração. Como Γ e Γ' atribuem o mesmo tipo para todo x livre em e e as regras do sistema de tipos são dirigidas por sintaxe, temos que Δ e Δ' são iguais. □

Apêndice B

Algoritmo para Generalização Mínima em Coq

Antes de descrever a formalização do algoritmo de generalização mínima em Coq, apresentaremos uma breve introdução ao assistente de provas Coq.

B.1 Introdução ao Assistente de Provas Coq

Coq é um assistente de provas baseado no cálculo de construções indutivas (CIC) [Bertot & Castéran, 2004], um λ-cálculo tipado de ordem superior, estendido com definições indutivas. A demonstração de teoremas em Coq é baseada nas idéias da chamada *correspondência BHK*[1], onde tipos representam fórmulas lógicas, λ-termos representam provas, e a tarefa de verificar se um termo é a uma prova de uma dada fórmula corresponde á tarefa de verificação de tipos [Sørensen & Urzyczyn, 2006].

Porém, escrever um termo cujo tipo corresponde a uma determinada fórmula lógica é tarefa árdua, mesmo para proposições simples. Visando simplificar a tarefa de provar teoremas, Coq provê *táticas*, que são comandos para auxiliar o desenvolvedor na construção de provas. A Figura B.1 apresenta um trecho de código Coq no qual são utilizados alguns recursos desta linguagem — tipos, funções e definições de provas. Este exemplo apresenta a definição do tipo de dados `nat` que representa números naturais em representação unária, utilizando os construtores `O` e `S`, que representam o número 0 e a operação de sucessor, respectivamente. A anotação `Set`, presente na definição do tipo `nat`, indica que este tipo pertence ao universo `Set`[2].

[1] Abreviação de correspondência de Brower, Heyting, Kolmogorov, de Bruijn e Martin-Löf. Essa correspondência é também denominada como o "Isomorfismo de Curry-Howard".

[2] A linguagem de tipos de Coq classifica todo tipo de dados utilizando universos. O universo `Set` é

```
Inductive nat : Set :=
   | O : nat  | S : nat -> nat.

Fixpoint plus (n m : nat) : nat :=
   match n with
      | O => m
      | S n' => S (plus n' m)
   end.

Theorem plus_O_r : forall n, plus n 0 = n.
Proof.
   intros n. induction n as [| n'].
   (**Case n = 0**) reflexivity.
   (**Case n = S n' **) simpl. rewrite -> IHn'. reflexivity.
Qed.
```

Figura B.1. Código Coq de Exemplo

O comando `Fixpoint` é utilizado para definir funções por recursão estrutural. Toda função em Coq deve ser total. Tal restrição é imposta para garantir que a lógica associada á linguagem Coq é consistente.

Além da possibilidade de declarar novos de tipos e funções, Coq permite a definição e prova de teoremas. A Figura B.1 apresenta um teorema simples sobre a função `plus`: para qualquer valor `n`, de tipo `nat`, `plus n 0 = n`. O comando `Theorem` permite a definição de uma fórmula lógica que desejamos demonstrar e inicia o *modo de construção interativa de provas*, onde táticas pode ser utilizadas para produzir um termo que corresponde á prova da fórmula em questão. Após o enunciado do teorema `plus_O_r`, é iniciada a construção interativa da prova deste teorema. Inicialmente temos que provar o seguinte objetivo:

```
 ============================
   forall n : nat, plus n 0 = n
```

Após o comando `Proof.`, podemos utilizar táticas para construir passo a passo um termo de um determinado tipo. A primeira tática utilizada na prova deste teorema é a tática `intros`, que é utilizada para mover premissas e variáveis universalmente quantificadas do objetivo para as hipóteses. Como resultado da aplicação da tática `intros` temos que a variável quantificada `n` foi movida do objetivo para as hipóteses, resultando na seguinte configuração:

```
 n : nat
============================
```

utilizado para definir tipos cujos termos representam valores (programas) e o universo `Prop` é utilizado para definir tipos cujos termos represem provas.

```
 plus n 0 = n
```

Provamos que `plus n 0 = n` por indução sobre a estrutura de `n`, usando a tática `induction`, que gera um objetivo para cada construtor do tipo `nat`. Após usar a tática `induction` temos que provar os seguintes objetivos:

```
2 subgoals

============================
   plus 0 0 = 0

subgoal 2 is:
 plus (S n') 0 = S n'
```

A igualdade `plus 0 0 = 0` é trivialmente verdadeira, pela definição da função `plus`. A tática `reflexivity` demonstra tais igualdades reduzindo ambos os lados de uma igualdade para formas normais. O próximo objetivo a ser provado é:

```
 n' : nat
 IHn' : plus n' 0 = n'
============================
   plus (S n') 0 = S n'
```

A tática `induction` gera automaticamente a hipótese de indução `IHn'` para este teorema. Para finalizar esta prova, devemos, de alguma maneira, transformar o objetivo em um equivalente que permita-nos utilizar a hipótese de indução. A tática `simpl` pode ser utilizada para realizar reduções no objetivo, com base na definição da função `plus`. Com isso temos:

```
 n' : nat
 IHn' : plus n' 0 = n'
============================
   S (plus n' 0) = S n'
```

Agora o objetivo possui como subexpressão o lado esquerdo da hipotése `IHn'` e, portanto, podemos utilizar a tática `rewrite` para reescrever uma hipótese de igualdade. Com isso, obtemos:

```
 n' : nat
 IHn' : plus n' 0 = n'
============================
   S n' = S n'
```

Finalmente, o objetivo `S n' = S n'` pode ser provado utilizando a tática `reflexivity`. O script de táticas para provar o teorema `plus_O_r` constrói o termo apresentado na

```
fun n : nat =>
   nat_ind
   (fun n0 : nat => n0 + 0 = n0) (eq_refl 0)
     (fun (n' : nat) (IHn' : n' + 0 = n') =>
      eq_ind_r (fun n0 : nat => S n0 = S n')
        (eq_refl (S n')) IHn') n
        : forall n : nat, n + 0 = n
```

Figura B.2. Termo que representa a prova do teorema `plus_0_r`.

Figura B.2, que é construído com o princípio de indução[3] para números naturais, `nat_ind`:

```
nat_ind
     : forall P : nat -> Prop,
       P 0 -> (forall n : nat, P n -> P (S n)) ->
       forall n : nat, P n
```

`nat_ind` espera como parâmetros uma properiedade sobre números naturais (um valor de tipo `nat -> Prop`), uma prova de que esta propriedade é valida para zero (um valor de tipo `P 0`) e uma prova de que se esta propriedade é válida para um número natural arbitrário `n`, então ela vale para `S n` (um valor de tipo `forall n : nat, P n -> P (S n)`). Além do uso de `nat_ind`, gerado pela tática `induction`, o termo da Figura B.2 usa o construtor do tipo que representa igualdades "`eq_refl`", criado pela tática `reflexivity`, e o termo `eq_ind_r` (gerado pela tática `rewrite`), que nos permite concluir que `P y` é verdadeiro sempre que `P x` e `x = y` são prováveis. Ao invés de usar táticas para provar teoremas, podemos construir manualmente termos do CIC para prová-los. Porém, mesmo para teoremas simples como `plus_0_r`, esta é uma tarefa complexa pois, exige um conhecimento detalhado do sistema de tipos do CIC.

Mesmo teoremas simples como `plus_0_r` necessitam de diversas táticas para serem provados. Visando facilitar a tarefa de provar teoremas complexos, Coq fornece algumas ferramentas para automação, a saber: combinadores de táticas, uma linguagem de domínio específico para definir táticas e táticas que implementam procedimentos de decisão para algum subconjunto específico de fórmulas lógicas. Apresentaremos essas características por meio do exemplo da figura B.3.

Combinadores de táticas (também chamados de *tacticals* ou táticas de ordem superior)[Bertot & Castéran, 2004] são táticas que recebem como parâmetro outras

[3]Para cada definição de tipos de dados, Coq gera automaticamente um princípio de indução. Detalhes sobre o processo de construção de tais princípios podem ser encontrados em [Bertot & Castéran, 2004, Capítulo 14]

táticas. O script de táticas do teorema `plu_0_r_1` usa dois combinadores de táticas: "`try`" e "`;`".

A tática `try T`, onde `T` é uma tática qualquer, comporta-se exatamente como `T`, exceto que, se `T` falha então `try T` deixa o objetivo inalterado, sem qualquer mensagem de erro.

A tática `T ; S` aplica a tática `S` a todos os objetivos gerados pela aplicação da tática `T` sobre o objetivo corrente. No teorema `plus_0_r_1`, a tática

```
induction n as [| n'] ; simpl ; try (rewrite IHn') ; reflexivity
```

usa o combinador ";" para aplicar `simpl ; try (rewrite IHn') ; reflexivity` aos dois objetivos gerado pela tática `induction`. A anotação "`as [| n']`" é um padrão de introdução[4] e é utilizado para especificar quais nomes de variáveis serão introduzidas em cada sub-objetivo. De maneira geral, padrões de introdução são especificados como uma lista de nomes entre colchetes, separados por "|".

Ainda considerando este exemplo, temos o uso da tática `rewrite` pelo combinador `try`. Este uso é necessário pois o primeiro objetivo, gerado pela tática `induction`, não possui uma igualdade (a hipótese de indução) que possa ser reescrita para finalizar a prova.

Em algumas situações, é interessante aplicar uma tática apenas quando o objetivo, ou hipóteses, possuem um determinado "formato". Coq provê uma linguagem de domínio específico para escrever táticas, denominada $\mathcal{L}$tac, que possui construções para realizar casamento de padrão sobre hipóteses e objetivos. Na Figura B.3 temos a definição de uma tática, denominada `simple_induction`, para provar o teorema `plus_0_r2`. Assim como na prova do teorema `plus_0_r1`, combinadores são utilizados. A definição de `simple_induction` utiliza a sintaxe de $\mathcal{L}$tac para casamento de padrão sobre hipóteses e objetivos. A construção `match goal with ... end` permite definir que a execução de uma tática seja condicionada ao formato de hipóteses e do objetivo atual. No exemplo, temos que a tática `rewrite H` é executada apenas se existir uma hipótese `H : ?x + 0 = ?x`, onde `?x` é uma variável que pode unificar com qualquer termo. Para cada componente que ocorre em uma construção `match goal`, o símbolo "`|-`" é utilizado para separar hipóteses (à esquerda) do objetivo (à direita).

Em Coq, existem diversas táticas capazes de provar automaticamente fórmulas pertencentes a algum subconjunto específico da lógica. Um exemplo deste tipo de tática é `auto`, que implementa um algoritmo de prova automática baseado em resolução, similar ao utilizado por Prolog. A tática `intuition` é utilizada para aplicar

[4]Tradução livre: *introduction pattern*.

```
Theorem plus_0_r_1 : forall n, plus n 0 = n.
Proof.
   intros n.
   induction n as [| n'] ; simpl ;
      try (rewrite IHn') ; reflexivity.
Qed.

Ltac simple_induction x :=
  induction x ; simpl ;
     try (match goal with
            | [H : ?x + 0 = ?x |- _ = _ ] => rewrite H
          end) ; auto.

Theorem plus_0_r2 : forall n, n + 0 = n.
Proof.
  simple_induction n.
Qed.

Theorem omega_ex
   : forall x y z t, x <= y <= z /\ z <= t <= x -> x = t.
Proof.
  intros x y z t H ; omega.
Qed.
```

Figura B.3. Código Coq de Exemplo

simplificações para fórmulas da lógica intuicionista e a tática `omega` implementa um procedimento de decisão para provar fórmulas da aritmética de Presburger, sem quantificadores. Maiores detalhes sobre táticas e sobre a linguagem de definição de táticas podem ser encontrados nos Capítulos 8 e 9 de [Team, 2012].

Um recurso interessante de Coq é a possibilidade de definir tipos de dados que combinem partes lógicas e valores. Tais tipos são conhecidos na literatura como *especificações fortes*[Bertot & Castéran, 2004], uma vez que permitem definir funções que produzem, não apenas um resultado, mas também uma prova de que este resultado possui alguma propriedade de interesse. Como um exemplo, considere o tipo `sig`, chamado de "tipo subconjunto", definido na biblioteca padrão de Coq como:

```
Inductive sig (A : Set) (P : A -> Prop) : Set :=
   | exist : forall x : A, P x -> sig A P.
```

O tipo `sig` é usualmente representado em Coq utilizando a seguinte sintaxe `{x : A | P(x)}`. O tipo `sig` possui um único construtor: `exist`, que possui dois parâmetros. O primeiro parâmetro `x`, de tipo `A`, representa o valor, e o segundo parâmetro, de tipo `P x`, denota o "certificado" de que o valor `x` tem a propriedade especificada pelo predicado `P`. Como exemplo, considere:

```
forall n : nat, n <> 0 -> {p | n = S p}.
```

Este tipo pode ser utilizado para especificar uma função que retorna o predecessor de um número natural `n`, junto com uma prova de que o valor retornado é predecessor de `n`. A definição de uma função de tipo `sig` requer a especificação do certificado. Assim como em teoremas, táticas podem ser utilizadas na definição de tais funções. Por exemplo, considere a seguinte definição de uma função que retorna o predecessor de um dado número natural, se este é diferente de zero:

```
Definition pred_certified : forall n : nat, n <> 0 -> {p | n = S p}.
   intros n H.
   destruct n as [| n'].
   (**Case n = 0**)
   destruct H. reflexivity.
   (**Case n = S n'**)
   exists n'. reflexivity.
Defined.
```

Outro exemplo de tipo que pode ser usado para especificações em Coq é o tipo `sumor`, definido na biblioteca padrão como:

```
Inductive sumor(A : Set) (B : Prop) : Set :=
   | inleft : A -> sumor A B | inright : B -> sumor A B
```

O tipo `sumor A B` pode ser escrito utilizando o seguinte açúcar sintático (ou, na terminologia de Coq, notação): `A + {B}`. Este tipo pode ser usado como o tipo de uma função que retorna um valor de tipo `A`, ou retorna uma prova de que alguma propriedade especificada por `B` é válida. Como exemplo, o seguinte tipo pode ser utilizado para especificar uma função que retorna o predecessor de um número natural, ou uma prova de que o número fornecido como parâmetro é igual a zero.

```
{p | n = S p} + {n = 0}
```

Visando facilitar a tarefa de construir funções cujos tipos envolvem especificações fortes, M. Souzeau desenvolveu a tática `Program`[Team, 2012, Sozeau, 2007], que permite a definição de funções sem a necessidade de especificar as partes correspondentes ao certificado lógico do resultado. Os componentes lógicos de uma função definida utilizando a tática `Program` podem ser especificados após sua declaração, provando *obrigações* geradas automaticamente. O próximo exemplo utiliza a tática `Program` para a definição de uma função.

```
Program Definition pred_cert (n : nat) : {p | n = S p} + {n = 0} :=
  match n with
    | 0 => inright _ | S p => inleft (exist _ p _)
  end.
  Next Obligation. auto. Defined.
  Next Obligation. auto. Defined.
```

Ocorrências do caractere "_" na definição de `pred_cert` representam termos que serão preenchidos quando da demonstração de obrigações de prova. Para a definição anterior, as seguintes obrigações (ambas provadas pela tática `auto`) são geradas:

```
Obligation 1 of pred_cert:
forall n : nat, 0 = n -> 0 = 0.

Obligation 2 of pred_cert:
forall n p : nat, S p = n -> S p = S p.
```

Podemos especificar que uma tática `T` seja aplicada automaticamente para resolver obrigações, utilizando o comando `Obligation Tactic := T`. No exemplo anterior, para resolver todas as obrigações, bastaria especificar a tática `auto` para ser aplicada automaticamente. Caso reste alguma obrigação a ser provada após a execução da tática especificada pelo comando `Obligation Tactic`, podemos utilizar o comando `Next Obligation` para provar a próxima obrigação restante.

O comando `Extraction pred_certified` descarta os componentes lógicos da função `pred_certified` e gera uma implementação desta em OCaml [Team., 2012], Haskell [Jones, 2002] ou Scheme [Dybvig, 2009]. O código OCaml gerado para `pred_certified` é apresentado a seguir.

```
(** val pred_cert : nat -> nat **)
let pred_cert = function
   | O -> assert false (* absurd case *)
   | S n0 -> n0
```

B.2 Cálculo da Generalização Mínima em Coq

Nesta seção, descrevemos uma formalização, em Coq, do algoritmo apresentado na Seção 5.2 para cálculo da generalização mínima de tipos simples. Para isso, devemos primeiramente definir um tipo em Coq para representar tipos simples. Utilizamos valores de tipo `nat` para representar identificadores para variáveis e construtores de tipos. Desta maneira, definimos do seguinte modo o tipo `ty` para representar tipos simples:

```
Inductive ty : Set :=
   | var : nat -> ty
   | con : nat -> ty
   | app : ty  -> ty -> ty.
```

Mapeamentos finitos (φ) entre pares de tipos e variáveis de tipos são representados por triplas contendo dois valores de tipo `ty`, e um número natural que representa um

identificador de uma nova variável. A seguinte definição é utilizada para representar mapeamentos finitos:

```
Definition phi := list (ty * ty * nat).
```

Mapeamentos finitos devem prover duas operações: inserir uma nova entrada em um mapeamento e verificar se existe uma entrada para um dado par de tipos. Uma vez que mapeamentos são representados como listas, a inserção de uma nova entrada é apenas o construtor "cons"[5] do tipo de dados lista.

A função `lookup_phi` implementa a operação de verificar se existe uma entrada associada a um par de tipos, em um dado mapeamento finito. Em caso positivo, o número natural que representa o identificador da variável de tipo associada a este par é retornado como resultado. O tipo de `lookup_phi` usa o predicado `In`, definido na biblioteca padrão de listas em Coq[6]. A definição de `lookup_phi` é feita utilizando a tática `Program`, apresentada na Seção B.1, que gera automaticamente obrigações de prova para garantir que a função em questão está de acordo com a especificação dada por seu tipo [Team, 2012, Sozeau, 2007].

Na Figura B.4 é apresentado o código da função `lookup_phi` que, dados dois tipos τ, τ' e um mapeamento φ, verifica se o domínio de `lookup_phi` contém o par (τ, τ'), isto é, se existe n tal que (τ, τ', n) ocorre na lista que representa φ. A definição de `lookup_phi` produz 8 obrigações (uma para cada equação que define esta função), que são resolvidas pela tática definida na Figura B.5. Esta tática realiza simplificações utilizando as táticas `program_simpl` e `intuition`, que simplificam definições feitas usando `Program` e aplicam todas as possíveis simplificações para fórmulas da lógica intucionista, sobre as hipóteses e objetivos, respectivamente. Na sequência, esta tática elimina quantificadores existenciais, disjunções, igualdades e contradições, que podem aparecer durante a execução desta tática sobre cada obrigação de prova.

Uma parte importante da implementação da função *lgen* em Coq é a criação de novas variáveis de tipos. Podemos considerar que uma variável de tipos é nova, se esta não ocorre previamente. Para implementar a geração de novas variáveis, seguimos a abordagem comumente utilizada por algoritmos de inferência de tipos: usar um contador[Peyton Jones et al., 2007]. Cada vez que o algoritmo necessita de uma nova variável, esta é gerada a partir do valor atual do contador, que é então incrementado para ser utilizado posteriormente.

Desta maneira, a definição da função *lgen* deve possuir como parâmetro adicional um número natural que será utilizado para a criação de novas variáveis de tipo. Este

[5]"cons" é o nome usual do construtor utilizado para inserir um novo elemento em uma lista.

[6]O predicado `In x l` é verdadeiro apenas se e somente se o elemento `x` pertence a lista `l`.

```
Program Fixpoint lookup_phi (t t' : ty) (l : phi) :
                  {n | In (t,t',n) l} + {~ exists n, In (t,t',n) l} :=
  match l with
    | nil => inright _
    | x :: l' =>
        match x with
          | (a,b,c) =>
              match eq_ty_dec t a, eq_ty_dec b t' with
                | left _, left _ => inleft _ (exist _ c _)
                | right _, left _ =>
                    match lookup_phi t t' l' with
                      | inleft x => inleft _ x
                      | inright _ => inright _ _
                    end
                | left _, right _ =>
                    match lookup_phi t t' l' with
                      | inleft x => inleft _ x
                      | inright _ => inright _ _
                    end
                | right _, right _ =>
                    match lookup_phi t t' l' with
                      | inleft x => inleft _ x
                      | inright _ => inright _ _
                    end
              end
        end
  end.
```

Figura B.4. Definição da função `lookup_phi`.

```
Obligation Tactic :=  program_simpl ; intuition ;
  repeat (match goal with
            | [H : ex _ |- _] =>
               let v := fresh "v" in destruct H as [v H]
            | [H : False |- _] => destruct H
            | [H : _ \/ _ |- _] => destruct H
            | [H : _ = _ |- _] => inverts* H end).
```

Figura B.5. Tática para resolver obrigações de prova de `lookup_phi`.

```
Program Fixpoint max_list (l : list nat) :
 {n | forall n', In n' l -> n > n'} :=
  match l with
    | nil => 0
    | x :: l' =>
        match max_list l' with
          | x' => match le_gt_dec x x' with
                    | left _ => S x'
                    | right _ => S x
                  end
        end
  end.
```

Figura B.6. Função `max_list`.

```
Obligation Tactic := program_simpl ; intuition ;
      repeat (match goal with
                | [H : _ \/ _ |- _] => destruct H ; subst
                | [H : context[In _ _ -> _],
                   H1 : In _ _ |- _] => apply H in H1 ; try omega
                     end).
```

Figura B.7. Tática para resolver obrigações de `max_list`.

parâmetro é devidamente inicializado, com um valor que é maior do que qualquer variável que já tenha sido utilizada previamente. Para obter este valor, utilizamos a função `max_list`, definida na Figura B.6, que retorna o sucessor do maior elemento de uma dada lista de números naturais.

Novamente, utilizamos tipos para codificar a especificação da função definida e todas as obrigações de prova são resolvidas pela tática apresentada na Figura B.7, que realiza simplificações e, na sequência, elimina disjunções e implicações que possuam o predicado `In` em seu lado esquerdo. Desigualdades aritméticas geradas são resolvidas pela tática `omega`, que é um procedimento de decisão para fórmulas da aritmética de Presburguer[Team, 2012].

O valor inicial do contador utilizado na função *lgen* para a criação de novas variáveis é o resultado de aplicar a função `max_list` sobre a lista de variáveis de tipo dos tipos de entrada da função *lgen*.

Para formalizar a função para o cálculo da generalização mínima de dois tipos, precisamos definir uma relação de ordem parcial sobre tipos simples. Ao invés de utilizarmos uma definição baseada em substituições (como apresentado na Seção 4.6.1.3), vamos usar um predicado recursivo, que captura a noção de quando um tipo é menor que outro. Este predicado é apresentado a seguir.

```
Inductive leq_ty : ty -> ty -> Prop :=
   | leq_var : forall n t, leq_ty (var n) t
   | leq_con : forall n, leq_ty (con n) (con n)
   | leq_app : forall l r l' r', leq_ty l l' ->
                                 leq_ty r r' ->
                                 leq_ty (app l r) (app l' r').
```

O construtor `leq_var` denota o fato de que uma variável é mais geral do que qualquer outro tipo simples e `leq_con` representa que um construtor de tipos pode ser somente a generalização de si próprio. Finalmente, uma aplicação `app l r` é mais geral que `app l' r'` se `l` é uma generalização de `l'` e `r` é mais geral que `r'`. A partir desta ordenação sobre tipos simples, podemos expressar o fato que um tipo `t1` é a generalização mínima de dois tipos simples `t` e `t'` utilizando a seguinte definição:

```
Definition least_gen (t t' t1 : ty) :=
   leq_ty t1 t /\ leq_ty t1 t' /\
      forall u, leq_ty u t /\ leq_ty u t' -> leq_ty u t1.
```

Na Figura B.8 temos a definição da função para cálculo da generalização mínima de dois tipos simples em Coq. Esta função recebe como parâmetros dois tipos, um mapeamento finito e um número natural, que representa a última variável gerada pelo algoritmo. Este número é utilizado para a criação de novas variáveis. A função `lgen_aux` retorna como resultado uma tripla, composta por um tipo, que é a generalização mínima dos tipos fornecidos como parâmetro, um mapeamento finito e um número natural, possivelmente atualizados. A definição desta função segue o algoritmo apresentado na Figura 5.2 e seu código é longo devido ao casamento de padrão exaustivo utilizado. Todas as obrigações de prova geradas pela função `lgen_aux` são provadas pela tática definida na Figura B.9, que expande a definição `least_gen` e realiza simplificações utilizando as táticas `intuition` e `program_simpl`. A função `lgen_aux` espera como parâmetro um número natural, que será utilizado para gerar novas variáveis. Para finalizar a definição do algoritmo para o cálculo da generalização mínima, devemos fornecer o valor adequado para este parâmetro de `lgen_aux`, que deve ser o resultado da função `max_list` aplicada ao conjunto de variáveis dos tipos fornecidos como parâmetro. A seguinte definição, construída utilizando táticas, realiza a tarefa de combinar estas funções.

```
Definition lgen (t t' : ty) : {t1 | least_gen t t' t1}.
   destruct (max_list (fv t ++ fv t')) as [v H].
   destruct (lgen_aux t t' nil v) as [[t1 [phi m]] Ht1].
   simpl in *. exists* t1.
Defined.
```

```
Program Fixpoint lgen_aux (t t' : ty) (l : phi) (m : nat) :
                    {p | least_gen t t' (@fst ty (phi * nat) p)} :=
  match t,t' with
    | var n, var n' =>
        match eq_nat_dec n n' with
          | left _ => exist _ (var n, (l, m)) _
          | right _ =>
              match lookup_gen_list (var n) (var n') l with
                | inleft (exist x _) => exist _ (var x, (l,m)) _
                | inright _ => exist _ (var m, ((var n, var n',m) :: l, S m)) _
              end
        end
    | var n, con n' =>
        match lookup_gen_list (var n) (con n') l with
          | inleft (exist x _) => exist _ (var x, (l,m)) _
          | inright _ => exist _ (var m, ((var n, con n',m) :: l, S m)) _
        end
    | var n, app f r =>
        match lookup_gen_list (var n) (app f r) l with
          | inleft (exist x _) => exist _ (var x, (l,m)) _
          | inright _ => exist _ (var m, ((var n,app f r,m) :: l, S m)) _
        end
    | con n, con n' =>
        match eq_nat_dec n n' with
          | left _ => exist _ (con n, (l,m)) _
          | right _ =>
              match lookup_gen_list (con n) (con n') l with
                | inleft (exist x _) => exist _ (var x, (l,m)) _
                | inright _ => exist _ (var m, ((con n,con n',m) :: l, S m)) _
              end
        end
    | con n, var n' =>
        match lookup_gen_list (con n) (var n') l with
          | inleft (exist x _) => exist _ (var x, (l,m)) _
          | inright _ => exist _ (var m, ((con n, var n',m) :: l, S m)) _
        end
    | con n, app f r =>
        match lookup_gen_list (con n) (app f r) l with
          | inleft (exist x _) => exist _ (var x, (l,m)) _
          | inright _ => exist _ (var m, ((con n, app f r,m) :: l, S m)) _
        end
    | app f r, con n =>
        match lookup_gen_list (app f r) (con n) l with
          | inleft (exist x _) => exist _ (var x, (l,m)) _
          | inright _ => exist _ (var m, ((app f r, con n,m) :: l, S m)) _
        end
    | app f r, var n =>
        match lookup_gen_list (app f r) (var n) l with
          | inleft (exist x _) => exist _ (var x, (l,m)) _
          | inright _ => exist _ (var m, ((app f r, var n,m) :: l, S m)) _
        end
    | app f r, app f' r' =>
        match lgen_aux f f' l m with
          | (t,(l',m')) =>
              match lgen_aux r r' l' m' with
                | (t', (l'', m'')) => exist _ (app t t', (l'', m'')) _
              end
        end
  end.
```

Figura B.8. Função para cálculo da generalização mínima.

```
Obligation Tactic :=
  unfold least_gen in * ; program_simpl ; intuition ;
      repeat (match goal with
                | [H : leq_ty _ _ |- _] => inverts* H
                | [H : ?n = ?n -> False |- _] => destruct H ; auto
              end).
```

Figura B.9. Táctica para provar as obrigações de prova de `lgen_aux`

.

Primeiramente, obtemos o sucessor do maior identificador de variáveis presentes nestes tipos e o chamamos de `v`. O valor `v` é fornecido como argumento para `lgen_aux`, que produz uma tripla formada por um tipo `t1`, um mapeamento finito `phi` e um número natual `m`. O primeiro componente desta tripla, `t1`, é o resultado desejado, que é então utilizado como argumento da tática `exists*`, que termina a definição de `lgen`, provando que esta definição está de acordo com a especificação dada por seu tipo.

Referências Bibliográficas

[Augustsson, 1998] Augustsson, L. (1998). Cayenne—a language with dependent types. Em *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.

[Bertot & Castéran, 2004] Bertot, Y. & Castéran, P. (2004). *Interactive Theorem Proving and Program Development, Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag.

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

[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 Haskell's Multi-Parameter Type Class Dilemma. *Proceedings of the Brazilian Symposium on Programming Languages*.

[Camarão et al., 2007] Camarão, C.; Vasconcellos, C.; Figueiredo, L. & ao, N. 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. Em *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. Em *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. Em *LFP '92: Proceedings of the 1992 ACM conference on LISP and functional programming*, pp. 170--181, New York, NY, USA. ACM.

[Committee, 2012] Committee, H. P. (September 2012). Haskell multiparameter type class dilemma.

[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.

[Dybvig, 2009] Dybvig, R. K. (2009). *The Scheme Programming Language*. MIT Press, fourth edição.

[Faxén, 2002] Faxén, K.-F. (2002). A static semantics for haskell. *J. Funct. Program.*, 12(5):295--357.

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

[Hall et al., 1996] Hall, C. V.; Hammond, K.; Peyton Jones, S. L. & Wadler, P. L. (1996). Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109--138.

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

[Jones, 1995a] Jones, M. P. (1995a). *Qualified types: theory and practice*. Cambridge University Press, New York, NY, USA.

[Jones, 1995b] Jones, M. P. (1995b). Simplifying and improving qualified types. Em *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pp. 160--169, 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. Em *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, 2002] Jones, S. P., editor (2002). *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/.

[Jones et al., 1997] Jones, S. P.; Jones, M. & Meijer, E. (1997). Type classes: exploring the design space. Em *Proceedings of the ACM Haskell Workshop*.

[Kiselyov et al., 2010] Kiselyov, O.; Jones, S. P. & Shan, C.-c. (2010). Fun with Type Functions. Em Roscoe, A. W.; Jones, C. B. & Wood, K. R., editores, *Reflections on the Work of C.A.R. Hoare*, History of Computing, chapter 14, pp. 301--331. Springer London, London.

[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., 2007] Peyton Jones, S.; Vytiniotis, D.; Weirich, S. & Shields, M. (2007). Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17(1):1--82.

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

[S. P. Jones and others, 2012] S. P. Jones and others (2012). GHC — The Glasgow Haskell Compiler Version 7.4.2. Users Guide. 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).

[Smith, 1991] Smith, G. (1991). *Polymorphic type inference for languages with overloading and subtyping*. PhD thesis, Cornell Univ.

[Sørensen & Urzyczyn, 2006] Sørensen, M. & Urzyczyn, P. (2006). *Lectures on the Curry-Howard Isomorphism*. Number v. 10 in Studies in Logic and the Foundations of Mathematics. Elsevier.

[Sozeau, 2007] Sozeau, M. (2007). Subset coercions in coq. Em *Proceedings of the 2006 international conference on Types for proofs and programs*, TYPES'06, pp. 237--252, Berlin, Heidelberg. Springer-Verlag.

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

[Stuckey & Sulzmann, 2005] Stuckey, P. J. & Sulzmann, M. (2005). A theory of overloading. *ACM Trans. Program. Lang. Syst.*, 27(6):1216--1269.

[Sulzmann et al., 2007] Sulzmann, M.; Chakravarty, M. M. T.; Jones, S. P. & Donnelly, K. (2007). System f with type equality coercions. Em *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. Em *In Proc. of APLAS'06*.

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

[Team, 2012] Team, C. D. (September 2012). The Coq proof assistant reference manual, version 8.4.

[Team., 2012] Team., I. O. (2012). Objective Caml (OCaml) programming language website. http://caml.inria.fr/.

[Voigtländer, 2008] Voigtländer, J. (2008). Asymptotic improvement of computations over free monads. Em *Proceedings of the 9th international conference on Mathematics of Program Construction*, MPC '08, pp. 388--403, Berlin, Heidelberg. Springer-Verlag.

[Volpano, 1994] Volpano, D. M. (1994). Haskell-style overloading is np-hard. Em *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. Em *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pp. 15--28, London, UK. Springer-Verlag.

[Vytiniotis et al., 2011] Vytiniotis, D.; Jones, S. L. P.; Schrijvers, T. & Sulzmann, M. (2011). Outsidein(x) modular type inference with local assumptions. *J. Funct. Program.*, 21(4-5):333–412.

[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.