RODRIGO GERALDO RIBEIRO

PROGRAMAÇÃO GENÉRICA USANDO O SISTEMA CT

Belo Horizonte, Minas Gerais
Dezembro de 2007

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Programa de Pós-Graduação em Cincia da Computao

PROGRAMAÇÃO GENÉRICA USANDO O SISTEMA CT

Dissertação apresentadoa ao Curso de Pós-Graduação em Cincia da Computao da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Cincia da Computao.

RODRIGO GERALDO RIBEIRO

Belo Horizonte, Minas Gerais
Dezembro de 2007

UNIVERSIDADE FEDERAL DE MINAS GERAIS

FOLHA DE APROVAÇÃO

Programação Genérica usando o Sistema CT

RODRIGO GERALDO RIBEIRO

Dissertação defendida e aprovada pela banca examinadora constituída por:

UFMG Carlos Camarão de Figueiredo – Orientador
Universidade Federal de Minas Gerais

UFOP Lucília Camarão de Figueiredo – Co-orientador
Universidade Federal de Minas Gerais

Ph. D Carlos Camarão de Figueiredo
Universidade Federal de Minas Gerais

Ph. D Lucília Camarão de Figueiredo
Universidade Federal de Minas Gerais

Belo Horizonte, Minas Gerais, Dezembro de 2007

Resumo

Na última década surgiram diversas abordagens para programação genérica em Haskell: PolyP, 'Scrap Your Boilerplate', Derivable Type-Classes, Generic Haskell, 'Generics for the Masses'etc.

A maioria dessas abordagens utiliza uma representação para a estrutura de definição de tipos algébricos, sendo funções genéricas definidas sobre esta representação estrutural e automaticamente instanciadas para tipos de dados definidos no programa. Nesse caso, instâncias de uma dada função genérica para diferentes tipos são definições sobrecarregadas da mesma função. SYB adota uma estratégia distinta, definindo uma biblioteca de funções genéricas que realizam travessias sobre valores de tipos de dados complexos. Tais funções genéricas podem ser usadas para definir funções que operam sobre componentes específicos de valores de tipos complexos, evitando que a definição dessas funções envolva código simples e repetitivo, responsável pela travessia de valores do tipo complexo (*boilerplate code*).

Neste trabalho, apresentamos uma descrição sucinta dessas diversas abordagens para programação genérica, assim como uma comparação entre as abordagens mais relevantes. Com base nessa análise, propomos uma abordagem para suporte a programação genérica em uma linguagem similar a Haskell, baseada no sistema de tipos CT. O sistema CT extende o sistema de Damas-Milner com suporte para sobrecarga, sem necessidade de declaração de classes de tipos, tal como em Haskell. O back-end do compilador desta linguagem foi implementado como parte deste trabalho. Além disso, foram identificadas duas extensões necessárias ao sistema ao sistema CT, para que possa prover suporte a programação genérica: polimorfismo de ordem superior e definição de *"funções polimórficas especializadas"*. Esta última extensão é baseada na idéia de definições sobrecarregadas sobrepostas para uma mesma função, que foi também implementada como parte deste trabalho.

Abstract

The last decade has seen a number of approaches to datatype-generic programming: PolyP, 'Scrap Your Boilerplate', Derivable Type-Classes, Generic Haskell, 'Generics for the Masses', etc. The approaches vary in sophistication and target audience: some propose fullblown programming languages, others suggest libraries.

Most of these approaches uses a structural representation of the definition of an algebraic data type, with generic functions defined over this structural representation and automatically instantiated for data types defined in the program. In this case, instances of a generic function for different types are overloaded. SYB adopts a different approach, by defining a library of generic combinators for the traversal of values of complex data types. These generic functions can be used to define functions that operate on specific components of these complex types, avoiding the boilerplate code involved on the traversal of the structure of values of these types.

This work presents a summary of these various approaches for generic programming and compares the most relevant ones. Based on this analysis, we propose an approach to support generic programming in a language similar to Haskell, based on type system CT — Haskell-CT. System CT extends the Damas - Milner type system with support for overloading, without the need of class declarations, as in Haskell. As a part of this work, a compiler for the language Haskell-CT has been implemented, by integrating system CT's front-end with the back-end of the Haskell compiler GHC. In addition, two extensions to system CT are identified in order to provide support for generic programming in this system: arbitrary-rank polimorphism and the definition of specialized polimorphic functions.

A meus pais, José e Maria,
a minha irmã, Fernanda e
a minha amada Dáfani.

Agradecimentos

Primeiramente, gostaria de agradecer a meus pais pelo seu apoio incondicional em meus estudos e pela presença e suporte que foram, sem dúvida alguma, fundamentais para a conclusão deste trabalho. À minha irmã, Fernanda, pelo bom humor.

Agradeço, também, ao Departamento de Ciência da Computação da Universidade Federal de Minas Gerais pela oportunidade de participar do seu programa de mestrado e à CAPES pelo apoio financeiro.

Agradeço o meu orientador, professor e amigo Carlos Camarão pela atenção, incentivo, disponibilidade e paciência diante da minha inexperiência e teimosia, que estiveram presentes em muitos momentos do desenvolvimento deste 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 Mariza Andrade da Silva Bigonha, Newton José Vieira e Nívio Ziviani, pelos ensinamentos nas disciplinas que tive oportunidade de cursar.

Ao amigo Denis Pinheiro, pelo companherismo desde a nossa graduação na UFOP.

Finalmente, gostaria de agradecer à sempre presente, doce e tão amada Dáfani, cuja presença nos momentos difíceis me deu força para continuar este trabalho.

Sumário

Lista de Figuras

Capítulo 1

Introdução

1.1 Motivação

Linguagens de programação modernas têm evoluído no sentido de utilizar sistemas de tipos mais flexíveis e com maior poder de expressão, evitando que programas que se comportem corretamente sejam rejeitados como incorretos, devido a restrições impostas pelo sistema de tipos. Tais sistemas de tipos têm o objetivo de prover suporte ao desenvolvimento de programas mais estruturados, aumentar a produtividade de programadores e ampliar a gama de erros detectados durante a compilação.

O *polimorfismo universal*, ou *polimorfismo paramétrico*, definido originalmente no sistema de tipos de *Hindley-Milner*, juntamente com o algoritmo de inferência de *Damas-Milner* [Hin69, DM82], constituem um marco histórico, por permitir expressar funções que podem operar sobre tipos de dados distintos de maneira segura, ou seja, garantindo que erros de tipo são sempre encontrados. O polimorfismo paramétrico está presente em diversas linguagens de programação modernas como Haskell[Jon03], Clean[PvENS], C++ (utilizando *templates*) [Str00b], Eiffel[Mey97] e Java, em sua versão mais recente.

O mecanismo de polimorfismo paramétrico é útil quando se deseja definir funções cujo comportamento independe do tipo dos elementos da estrutura de dados manipulada. De fato, grande parte das funções que os programadores desenvolvem repetidamente em bibliotecas de manipulação de estruturas de dados são funções polimórficas. Por exemplo, calcular o número de componentes de uma estrutura de dados, "filtrar"(descartar) todos os componentes que não satisfazem a um dado predicado, aplicar uma dada função a todos os componentes de uma estrutura de dados etc.

Entretanto, em grande parte das vezes, deseja-se definir funções que não operam da mesma maneira sobre valores de qualquer tipo, mas têm comportamento distinto, embora análogo, para argumentos de cada um desses tipos. Exemplos comuns são funções

para comparar, para transformar em cadeia de caracteres e imprimir (*pretty-printers*), para ler (*parsers*) etc. Funções dessa natureza podem ser mais facilmente definidas em linguagens que provêem suporte a *sobrecarga* de definições de funções, o que comumente é chamado de *polimorfismo ad-hoc*. Combinado com polimorfismo paramétrico, no entanto, resulta no que gostaríamos de chamar de *polimorfismo completo*, ou *refinado*, o qual não distingue apenas monomorfismo de um polimorfismo *"para todos"* (universal), mas possibilita também um polimorfismo *"para alguns"* (todos os que são válidos, dado um contexto). Vamos ser modestos e chamar esse tipo de polimorfismo apenas de *polimorfismo de sobrecarga* ou *polimorfismo universal com restrições*.

A linguagem Haskell utiliza um sistema de tipos que estende o sistema de Damas-Milner com *classes de tipos* para permitir a verificação estática e um tratamento uniforme a sobrecarga. Apesar do sistema de classes de tipos representar um grande passo para um suporte efetivo à sobrecarga juntamente com polimorfismo e inferência de tipos, este possui a desvantagem de obrigar o programador a criar declarações de classes para definições de símbolos sobrecarregados e requer, nessas classes, que os tipos destes símbolos sejam explicitamente anotados. Como exemplo, considere a definição das funções de projeção *fst* e *snd*, mostradas na Figura 1.1, sobrecarregadas para duplas e triplas. Em Haskell, para a definição destes dois símbolos houve a necessidade de declarar duas classes, uma relativa ao símbolo *fst* e outra para o símbolo *snd*. Mesmo considerando que *Haskell* permite que diversos símbolos sejam declarados em uma classe (reduzindo, assim, o número de classes necessárias), nem sempre a relação entre os tipos permite a utilização deste recurso. No exemplo mostrado na Figura 1.1, a relação entre os tipos impede que as funções sejam declaradas como membros da mesma classe.

O sistema de tipos *CT*[1][CaLFN07, CF99b] também estende o sistema de Damas-Milner com suporte a sobrecarga, porém, ao contrário do sistema utilizado na linguagem Haskell, este não obriga o programador a declarar classes para definir símbolos sobrecarregados, assim como não obriga o programador a anotar o tipo desses símbolos. Na Figura 1.1 é mostrado o trecho de código que define os símbolos *fst* e *snd* para duplas e triplas em uma linguagem que utiliza o sistema *CT*. Como pode ser observado, não houve a necessidade de quaisquer declarações adicionais para a definição destes símbolos, uma vez que o sistema *CT* é capaz de inferir os tipos destes.

Apesar do suporte ao polimorfismo paramétrico e sobrecarga em linguagens de programação modernas serem características úteis para o densenvolvimento de programas, tem-se observado, nesta última década, um crescente interesse pela chamada *programação genérica*. Entretanto, não existe uma única definição para este termo.

[1] *Constrained Types*

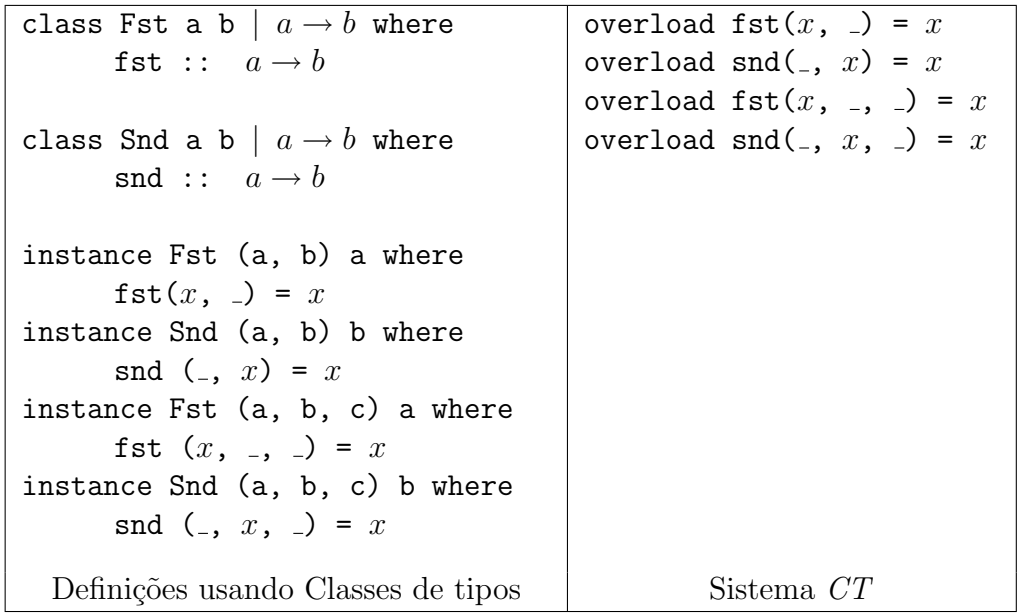

Figura 1.1: Definições sobrecarregadas de funções de projeção em *Haskell* e em uma linguagem baseada no Sistema *CT*.

Em linguagens como C++ e Java, o termo programação genérica é utilizado como um sinônimo de polimorfismo paramétrico. Em linguagens funcionais, este termo é utilizado como sinônimo do chamado polimorfismo estrutural, que permite definir funções cujo comportamento varia de acordo com a estrutura algébrica do tipo dos dados que são manipulados. Neste trabalho, o termo programação genérica será utilizado como sinônimo de polimorfismo estrutural, seguindo a terminologia de linguagens funcionais.

Existem atualmente diversas propostas de linguagens e bibliotecas para programação genérica em Haskell [HJL06, AP01, JBM98, AM03]. A maioria dessas abordagens utiliza uma representação para a estrutura de definição de tipos de dados algébricos, sendo as funções genéricas definidas na forma de uma análise de casos sobre a estrutura algébrica de tipos e automaticamente instanciadas para tipos de dados definidos no programa, conforme requerido. Nesse caso, instâncias de uma dada função genérica para diferentes tipos são definições sobrecarregadas de mesma função, sendo a sobrecarga definida por meio do mecanismo de *Type Classes*. Outra possível abordagem [HJL06] é definir uma biblioteca de funções genéricas que realizem travessias sobre valores de tipos de dados complexos, e que possam ser usadas para definir funções que operam sobre componentes específicos de valores desses tipos, evitando a tediosa repetição do código responsável pela travessia de valores do tipo complexo (*boiler-plate code*). Nesse caso, funções que operam sobre um componente específico do tipo complexo são estendidas de modo a operar sobre quaisquer componentes desse tipo, utilizando um mecanismo de coersão de tipos. Também nesse caso é o mecanismo de

classes de tipos de Haskell que é usado, para definir funções sobrecarregadas de coersão de tipos.

Com base em uma análise dessas diferentes abordagens, buscamos, neste trabalho, responder às seguintes perguntas: Como o suporte a sobrecarga oferecido pelo sistema *CT* pode ser útil no suporte a um estilo de programação genérica baseado na definição de funções que podem operar sobre construtores de tipos distintos? Como esse estilo se relaciona com as demais abordagens? Quais são as extensões necessárias ao sistema de tipos CT para prover suporte a programação genérica?

O objetivo deste trabalho é estudar como o sistema CT pode ser usado como base para programação genérica. Mais especificamente, o trabalho visa identificar extensões necessárias a este sistema para um suporte adequado a programação genérica, baseado no uso de sobrecarga. Visamos também uma integração da implementação do algoritmo de inferência de tipos disponível para o sistema CT com o back-end de um compilador Haskell, de modo a poder realizar testes que contenham exemplos de funções definidas com base no estudo acima referido.

1.2 Contribuições da dissertação

As principais contribuições deste trabalho são:

- Identificação de extensões necessárias ao sistema CT para suporte a programação genérica de maneira adequada: sem necessidade de realização de conversões de tipos que podem ocasionar erros durante a execução de programas, sem a necessidade de definir funções baseadas em indução estrutural e sem a necessidade de uso de código *boilerplate* (Capítulo 5).

- Apresentação das principais abordagens existentes para Programação Genérica em Haskell e uma comparação entre as abordagens mais relevantes (Capítulo 4).

- Integração do *front-end* implementado em [Vas04] com o *back-end* do compilador Haskell GHC (Capítulo 6). Como resultado desse trabalho obtivemos o primeiro compilador para uma linguagem baseada no sistema CT.

- Definição e implementação de uma extensão da política de sobrecarga do Sistema CT para permitir definições de símbolos sobrecarregados com tipos sobrepostos (Capítulo 5).

1.3 Organização da dissertação

O restante deste trabalho está organizado da seguinte maneira: Capítulo 2 apresenta uma breve introdução à linguagem Haskell, descrevendo algumas características e recursos desta linguagem. Capítulo 3 apresenta duas abordagens para polimorfismo de sobrecarga. A primeira é utilizada pela linguagem Haskell, na qual o programador define símbolos sobrecarregados por meio da declaração de classes e definição de instâncias dessas classes, o que provê uma abordagem mais flexível do que a normalmente usada em linguagens de programação tradicionais. A segunda abordagem é a utilizada pelo sistema CT, que possibilita a definição de símbolos sobrecarregados sem a necessidade de declaração de classes de tipos. Para ambas as abordagens é apresentada uma descrição informal, incluindo o processo de detecção de ambigüidades. Capítulo 4 apresenta diferentes abordagens para programação genérica em Haskell, incluindo exemplos canônicos de funções genéricas, para ilustrar cada uma das abordagens. Ao final desse capítulo é apresentada uma análise comparativa entre *Generic Haskell* e *Scrap your boilerplate*. Capítulo 5 apresenta extensões para um suporte adequado a programação genérica, baseada no sistema CT, explicando com exemplos ilustrativos porque essas extensões são necessárias. Capítulo 6 apresenta detalhes da integração do *front-end* implementado em [Vas04] com o *back-end* do compilador Haskell GHC. Capítulo 7 conclui o trabalho e aponta possíveis trabalhos futuros.

Capítulo 2

A Linguagem Haskell

Este capítulo apresenta uma breve introdução à linguagem Haskell. Leitores familiarizados com esta linguagem podem continuar a leitura a partir do Capítulo 3.

"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 arbitária e números de ponto flutuante. Haskell é o ápice da solidificação de vários anos de pesquisa em linguagens funcionais não-estritas"(O relatório Haskell [Jon03]).

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

2.1 Módulos

Programas em Haskell são compostos por um conjunto de **módulos**. Módulos provêem uma forma para o programador re-utilizar código e controlar o espaço de nomes em grandes programas. Cada módulo é composto por um conjunto de *declarações*, que podem ser: declarações de classes, instâncias, tipos de dados, valores, sinônimos de tipos, funções, etc. A Figura 2.1 mostra o trecho de código correspondente ao módulo `Table` que implementa uma tabela por meio de uma lista de pares chave-valor. Este módulo define a constante `emptyTable` e as funções `insertTable`, `elemTable`, `searchTable`, `removeTable` e `updateTable` para manipulação desta tabela.

2.2 Anotações de Tipo

No módulo `Table`, cada definição de função / constante é precedida por sua correspondente *anotação de tipo*. Uma anotação de tipo fornece uma especificação parcial do significado de um símbolo.

A linguagem Haskell, por ser baseada no sistema de tipos de Hindley-Milner, permite o uso de polimorfismo paramétrico. Todos os símbolos definidos no módulo `Table` são *polimórficos*. Por exemplo, a constante `emptyTable`, possui o tipo `Table a`, que corresponde a um sinônimo para o tipo `[(String, a)]`, que indica que este símbolo pode assumir diferentes tipos de acordo com o valor da *variável de tipo* `a`; caso `a` assuma o valor `Int`, este tipo seria `[(String, Int)]`; se `a` assumir o valor `Bool`, então este tipo seria `[(String, Bool)]`, etc.

Evidentemente, a linguagem Haskell também suporta a definição de tipos *monomórficos*. A constante `pi`, definida abaixo, possui o tipo *monomórfico* `Double`:

```
pi :: Double
pi = 3.1415
```

Tipos funcionais especificam os tipos dos parâmetros e do resultado de uma função. O símbolo `searchTable` possui a seguinte anotação de tipo: `String → Table a → a`, que especifica que esta função recebe como parâmetros um valor do tipo `String` e uma lista de pares compostos por uma `String` e um elemento de um tipo qualquer e retorna como resultado um elemento deste tipo.

Cabe ressaltar que, com poucas exceções, anotações de tipos são opcionais em programas Haskell, uma vez que, o compilador é capaz de inferir um tipo para cada símbolo, que representa todo o conjunto dos possíveis tipos que o programador poderia anotar para este. Este processo de determinar o tipo para um símbolo não anotado é chamado de *inferência de tipos*. Caso o programador forneça uma anotação de tipo para um símbolo, o compilador verifica se a definição dada para o símbolo pode ter o tipo anotado. Este processo de verificação é chamado *verificação de tipos*.

2.3 Sintaxe de Listas

Listas são estruturas de dados usadas comumente para modelar diversos problemas. Por isto, a linguagem Haskell possui uma sintaxe especial para representar este tipo de dados. O tipo de dados lista, `[a]`, pode ser definido indutivamente como uma lista vazia, representada por `[]`, ou por uma lista `x : xs` contendo um primeiro elemento `x`, seguido de uma lista `xs`. Os símbolos `[]` e `:` são *construtores de dados* do tipo lista, cujos tipos são respectivamente `[a]` e `a → [a] → [a]`.

```
module Table where


type Table a = [(String, a)]

emptyTable :: Table a
emptyTable = []


insertTable ::  String → a → Table a → Table a

insertTable s a t
                | elemTable s t = t
                | otherwise = (s, a) : t


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


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


updateTable ::  String → a → Table a → Table a

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


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

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

Figura 2.1: Trecho de um programa Haskell

Uma primeira forma de sintaxe especial para listas é a utilizada na constante:

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

que corresponde a uma abreviação para

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

No módulo `Table`, a função `elemTable` usa outro tipo de sintaxe especial para listas, que é baseada na notação de teoria de conjuntos. Esta função poderia ser representada em teoria de conjuntos como:

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

O último tipo de *açúcar sintático* fornecido pela linguagem Haskell para listas é utilizada para facilitar a definição de seqüências aritméticas:

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

2.4 Casamento de Padrões

O casamento de padrões desempenha um papel fundamental nas definições de funções em linguagens funcionais modernas, por meio de equações. A função `removeTable`, 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, onde cada uma especifica o padrão da lista recebida como argumento. A primeira equação utiliza o padrão `[]` para representar listas vazias, enquanto a segunda equação utiliza o padrão `(x:xs)` que casa com listas que contêm pelo menos um elemento.

A linguagem Haskell possui dois tipos especiais de descrições de padrões. O primeiro permite o casamento de padrão não estrito. O símbolo "~"precedendo um padrão faz com que este padrão seja avaliado somente quando este for necessário, tornando-o irrefutável. Por exemplo, considere a seguinte definição:

```
g ::  Bool → (Int, Int) → Int
g b ~(x,y) = if b then x + y else 0
```

Neste exemplo, o padrão correspondente à tupla `(x,y)` foi marcado como sendo não-estrito. Portanto, os valores de `x` e `y` somente serão ligados quando `b` for verdadeiro, ou seja, quando estes forem necessários.

```
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.

Outro de tipo de definição de padrão - $n + k$ - é utilizado para definir padrões para tipos numéricos, tal como `(n + 2)` no seguinte exemplo de cálculo de um termo da série de fibonacci:

```
fib ::  Int → Int
fib 0 = 0
fib 1 = 1
fib (n + 2) = fib n + fib (n + 1)
```

Guardas

A definição da função `insertTable` é um exemplo de definição que utiliza *expressões com guardas*, que permitem a definição de alternativas para uma mesma equação. A alternativa a ser executada é selecionada de acordo com a avaliação das expressões booleanas (guardas) especificadas.

2.5 Tipos de Dados

A seguir, nas Figuras 2.2 e 2.3 são mostradas declarações de um tipo de dados algébrico e de uma função que recebe este tipo como argumento, que ilustram as características básicas de tipos de dados algébricos em Haskell.

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

Tipos de dados algébricos em Haskell consistem de uma *soma de produtos*. A definição do tipo de dados `Tree a` é uma folha (`Leaf`) que corresponde a um produto trivial contendo somente um tipo, e um nodo(`Node`) contendo uma sub-árvore a sua esquerda e outra à direita, que corresponde a um produto contendo um elemento do

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

Figura 2.3: Tipo de dados algébrico.

```
data Seq a = Nil | Cons a (Seq (a, a))
```

Figura 2.4: Tipo de dados algébrico não regular.

```
length ::  Seq a → Int
length Nil = 0
length (Cons x s) = 1 + 2 * (length s)
```

Figura 2.5: Função definida sobre um tipo de dados não regular.

tipo `a` e dois elementos de tipo `Tree a`. O tipo de dados `Tree a` corresponde a soma de dois produtos, um correspondente ao construtor de dados `Leaf` e o construtor `Node`.

O tipo de dados `Tree a` é um exemplo dos chamados *tipos de dados regulares*, i.e tipos de dados que possuem um componente recursivo idêntico ao tipo sendo definido. Porém, em diversas situações é interessante a utilização de *tipos de dados não regulares*. Para estes tipos de dados, o componente recursivo assume um tipo diferente do declarado. Um exemplo é o tipo de árvores binárias perfeitamente balanceadas [Oka98] mostrado na Figura 2.4:

Neste exemplo, o componente recursivo `Seq (a,a)` possui um tipo diferente do tipo `Seq a` que está sendo definido.

Tipos de dados não regulares são importantes por capturarem invariantes que tipos de dados regulares não são capazes de representar. Funções definidas sobre tipos de dados não regulares, muitas vezes, são mais eficientes que suas correspondentes uniformes. Por exemplo, a função `length`, definida na Figura 2.5, calcula o número de elementos presentes em uma estrutura do tipo `Seq a` em tempo $O(\log\ n)$.

2.6 Conclusão

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

Capítulo 3

Polimorfismo de Sobrecarga

Strachey, em 1964, foi o primeiro a utilizar o termo *polimorfismo ad-hoc* para se referir a funções polimórficas que podem ser aplicadas a argumentos de diferentes tipos, mas que comportam-se de acordo com o tipo do argumento para as quais são aplicadas. Neste texto, será usado, preferencialmente, o termo *polimorfismo de sobrecarga* para denominar este tipo de função. Linguagens que provêem suporte ao polimorfismo de sobrecarga permitem fazer várias definições, todas com o mesmo nome. A tarefa de determinar qual função é chamada pode ser realizada estaticamente pelo compilador, que toma esta decisão com base em informações do contexto onde o nome da função é usado, ou pode ser realizada dinamicamente, ou seja, durante a execução do programa [Str00a].

Ao contrário do *polimorfismo paramétrico*, a importância do polimorfismo de sobrecarga é muitas vezes subestimada, considerando que este não aumenta a expressividade de uma linguagem, pois poderia ser eliminado por uma renomeação adequada de símbolos. Todavia, a importância do polimorfismo de sobrecarga não está em evitar a poluição do espaço de nomes, mas na propriedade de que expressões e nomes definidos utilizando símbolos sobrecarregados podem ser usados em contextos que podem requerer valores de tipos distintos [CF99a].

Linguagens que provêem polimorfismo de sobrecarga utilizam uma *política de sobrecarga* (*overloading policy*), que estende a possibilidade de sobrecarga para permitir que um maior número de programas que utilizam símbolos sobrecarregados sejam considerados corretos e, ao mesmo tempo, estabelece regras que limitam a sobrecarga, para permitir que o processo de inferência de tipos seja eficiente. Uma estratégia de sobrecarga pode ser caracterizada como *dependente de contexto* ou *independente de contexto* [Wat90]. Em uma estratégia de sobrecarga independente de contexto, se f é um símbolo sobrecarregado então, para cada aplicação $f\ e$, a decisão sobre qual função f será aplicada é determinada de acordo com o tipo da expressão e. Por sua vez, uma

estratégia de sobrecarga dependente de contexto pode utilizar o contexto no qual $f\ e$ é usada para determinar qual definição do símbolo f será aplicada.

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

Muitos sistemas de tipo que suportam polimorfismo de sobrecarga têm adotado uma estratégia dependente de contexto para sobrecarga, por esta ser menos restritiva. Nesta classe de sistemas de tipos estão incluídos o sistema *CT*[CF99a, Vas04, CaLFN07] e o sistema de classes tipos utilizado pela linguagem *Haskell*[Jon03, WB89]. No restante deste capítulo serão apresentadas estas duas abordagens para o polimorfismo de sobrecarga. A primeira é a utilizada pela linguagem Haskell e a segunda é utilizada no compilador experimental desenvolvido como parte deste trabalho, que utiliza como sistema de tipos o sistema *CT*.

3.1 Polimorfismo de Sobrecarga em Haskell

3.1.1 Classes de tipo

Classes de tipo, em Haskell[Jon03], permitem ao programador definir símbolos sobrecarregados e seus respectivos tipos, que podem ser usados então para diferentes tipos, definidos como instâncias de uma classe.

Especifica-se que um determinado tipo pertence a uma classe por meio de uma declaração de *instância* desta classe. Uma declaração de instância de uma determinada classe fornece a definição para os símbolos desta classe, para um tipo específico. Como um primeiro exemplo (baseado em um exemplo de [HHJW07]), considere a classe `Eq`, mostrada na Figura 3.1, e suas instâncias para `Int` e `Bool`.

Suponha que `primEqInt` seja uma função primitiva, de tipo `Int` → `Int` → `Bool`, que verifique a igualdade de dois números inteiros. Considerando as duas instâncias

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

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

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

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

Figura 3.1: Exemplo de classe de tipos e instâncias

definidas para a classe `Eq`, tem-se que as seguintes expressões `2 == 3` e `False /= False` são bem tipadas. De maneira similar, a seguinte declaração polimórfica também é bem tipada:

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

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

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

Uma interessante característica de funções sobrecarregadas definidas por classes de tipo é que estas podem ser traduzidas em funções não sobrecarregadas equivalentes recebendo um argumento extra, denominado dicionário [WB89], que armazena, para cada tipo que é instância de uma classe, funções que definem, para este tipo, os símbolos da classe. O trecho de código na Figura 3.3 apresenta o resultado da tradução do código apresentado na Figura 3.1 para o equivalente utilizando dicionários.

Pode-se observar, na tradução mostrada na Figura 3.3, que a declaração de classe foi convertida para uma definição de um novo tipo de dados, que representa o dicionário para a classe `Eq`. Este tipo de dados possui como único construtor de dados,

```
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 3.3: Exemplo de tradução de classes de tipo e instâncias para dicionários.

`MkEq`, que possui como parâmetros dois valores do tipo funcional `a → a → Bool`, que correspondem às funções membro `(==) e (/=)`.

Cada uma das instâncias, definidas na Figura 3.3, foi traduzida para uma função que pode receber dicionários como argumentos e retorna dicionários mais complexos. A instância que implementa a igualdade para listas, após a tradução, recebe como parâmetro adicional um dicionário para representar as operações da classe `Eq` de seus elementos.

A tradução da função `member` (Figura 3.4), definida originalmente na Figura 3.2, possui um parâmetro extra correspondente ao dicionário da classe `Eq`, que representa a restrição `Eq a` presente em seu tipo original. Além disto, a utilização do símbolo sobrecarregado `(==)` é substituída pela função de projeção `eq`, conforme pode ser observado:

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

Figura 3.4: Tradução da função `member`, utilizando dicionários.

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

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

Figura 3.5: Exemplo de hierarquia de classes de tipos.

Na Figura 3.5, a classe `Ord` é definida como *subclasse* de `Eq`. Assim sendo, um tipo somente pertencerá a classe `Ord` se este já pertencer a classe `Eq`. A formação de hierarquia de classes pode simplificar os tipos de expressões envolvendo símbolos sobrecarregados, como no seguinte exemplo:

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

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

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

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

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

3.1.2 Ambigüidade

Um problema inerente ao polimorfismo de sobrecarga dependente de contexto é a possibilidade de ocorrência de *ambiguidades*. Considere o seguinte exemplo clássico [Jon03, HHJW07]:

```
show :: Show a ⇒ a → String
read :: Read a ⇒String → a
f :: String → String
f s = show (read s)
```

Neste exemplo, `show` converte um valor de qualquer tipo pertencente a classe `Show` para uma `String`, enquanto `read` faz o inverso para qualquer tipo da classe `Read`. Aparentemente, `f` é bem tipada, mas na verdade não é. O que faz com que `f` não seja bem tipada é o tipo da expressão intermediária `(read s)`. Para entender o motivo, suponha que os tipos `Int`, `Bool` e `String` possuam instâncias definidas das classes `Read` e `Show`, no contexto onde ocorre a definição de `f`. Uma vez que esta classe possui estas três instâncias e o tipo de `read` é $\forall$`a.{Read a}` $\Rightarrow$ `String` $\rightarrow$ `a` não é possível determinar qual o tipo que a variável `a` irá assumir, pois esta pode ser instanciada como `Int`, `Bool` ou `String`. Tais expressões, em Haskell, são ditas ambíguas e são rejeitadas como um erro de tipo. A linguagem Haskell possui a seguinte forma geral de tipos: $\forall \overline{u}.cx \Rightarrow t$, onde $\overline{u}$ é um conjunto de variáveis de tipo e cx são as restrições destas variáveis em relação ao tipo t. A partir desta forma geral, pode-se dizer que o tipo de uma expressão é ambíguo se este contém alguma variável de tipo presente nas restrições (cx) que não está presente no tipo (t). Diz-se que tal ocorrência desta variável, neste tipo, é ambígua. Como o tipo inferido para `show(read s)` é $\forall a.${`Read a, Show a`}`.String`, este é considerado ambíguo e será rejeitado pelo compilador.

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

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

Com a anotação de tipo presente na subexpressão `(read s)::String`, a definição do símbolo `f` torna-se não ambígua.

Ambiguidades em operações da classe `Num`[1] são muito comuns. Para evitar a necessidade de tipar toda subexpressão numérica, Haskell adota uma regra bastante *ad hoc*, explicada a seguir, para permitir a eliminação de algumas ambigüidades, 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 ambigüidade, uma variável de tipo `v`, é instanciável de forma a eliminar a ambigüidade, se:

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

[1] A classe `Num` representa tipos númericos em Haskell.

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

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

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

3.1.3 Polimorfismo de Sobrecarga e de Ordem Superior

Uma das primeiras extensões propostas para o sistema de classes de tipos de Haskell é de autoria de Mark Jones. Em [Jon93], sugere-se que uma classe de tipos poderia ser parametrizada sobre um *construtor de tipos* ao invés de um tipo, uma idéia denominada de *classes de construtores*. Uma aplicação imediata desta idéia era a definição de uma classe para representar uma *mônada*[Wad92]:

```
class Monad m where
    return :: a → m a
    (>>=) :: m a → (a →m b) → m b
```

Figura 3.6: Classe de tipo para representar uma mônada.

Neste exemplo (Figura 3.6), a variável de tipo `m` possui o *kind* [2] $\star \rightarrow \star$, sendo assim, `m` pode ser instanciada para um construtor de tipos, como no exemplo a seguir:

Na figura 3.7, instanciando o tipo do símbolo `return` (`a → m a`) de forma tal que `m = Maybe` tem-se exatamente o tipo de `return` na declaração da instância desta classe

[2] *Kinds* classificam tipos da mesma maneira que tipos classificam valores. O kind $\star$ é dito ser o kind de "tipos", portanto, se `m` possui o kind $\star \rightarrow \star$, então `m` mapeia um tipo em outro.

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
   return = Just
   Nothing >>= k = Nothing
   (Just x) >>= k = k x
```

Figura 3.7: Instância da classe `Monad`

para o tipo `Maybe`. Com a popularização do sistema de I/O monádico, a utilização de classes de construtores para a definição de símbolos polimórficos de ordem superior tornou-se padrão [HHJW07].

Todavia, a utilidade do polimorfismo de ordem superior não se restringe à definição de classes de tipos. Uma importante aplicação deste recurso é a definição de tipo de dados parametrizados sobre variáveis de tipo de *kind* superior, como:

```
data GRose f a = GNode a (f (GRose a))
```

Além disso, o polimorfismo de ordem superior é fundamental para a definição de funções sobre tipos de dados não regulares[BP99, Oka99].

3.1.4 Classes de Tipo com Múltiplos Parâmetros

A generalização do conceito de classes de tipo, permitindo a definição dessas classes com múltiplos parâmetros, foi originalmente proposta em [WB89]. Considere o seguinte exemplo (transcrito de [Jon00]):

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

Enquanto uma classe de tipos com um único parâmetro pode ser vista como um predicado sobre tipos (por exemplo, o predicado `Eq a` é válido sempre que a operação de igualdade é definida para o tipo `a`), uma classe de tipos com múltiplos parâmetros pode ser vista como uma relação entre tipos[3] (por exemplo, a relação `Coerce a b` é válida sempre que existir uma função `coerce :: a → b`).

Diversos artigos incluem exemplos envolvendo classes de tipo com múltiplos parâmetros [JJM97, DO02]. Embora várias implementações de Haskell incluam suporte a classes com múltiplos parâmetros, essa extensão ainda não foi incluída na definição oficial da linguagem, por requerer um tratamento adicional para evitar a ocorrência de ambigüidades. Para ilustrar este fato, considere o seguinte exemplo, transcrito de [Jon00]:

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

[3]Qualquer relação $R \subseteq A \times B$ pode ser vista como um predicado $p : A \times B \rightarrow Bool$

O exemplo anterior, descreve uma classe de tipos para representar operações sobre coleções, onde a variável `a` representa o tipo dos elementos e `b` a coleção em si. Pode-se definir como instâncias da classe `Collects`:

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

Possíveis instâncias para esta classe seriam:

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

Neste exemplo ocorre um problema com o tipo da função `empty`. De acordo com a regra de ambigüidade adotada em Haskell (apresentada na Seção 3.1.2), essa função é considerada ambígüa, uma vez que seu tipo é `empty :: {Collects a b}. b`.

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

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

Apesar desta declaração não possuir o problema de ambigüidade em relação ao símbolo `empty`, esta apresenta o inconveniente de poder ser instanciada apenas para coleções formadas por um construtor de tipos `c` aplicado a um tipo `a`. Para resolver estes problemas de ambigüidades inerentes ao uso de classes de múltiplos parâmetros, em [Jon00] foi proposta uma extensão ao mecanismo de classes de tipos denominado *dependências funcionais*, que será apresentado na próxima seção.

3.1.5 Dependências Funcionais

A utilização de classes de tipo com múltiplos parâmetros, apesar de ser uma extensão útil para a definição de símbolos sobrecarregados, facilita a declaração de tipos ambígüos. Como exemplo (transcrito de [HHJW07]), 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 Int Float where ...
instance Add Int Float Float where ...
instance Add Float Int Float where ...
instance Add Float 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. Apesar desta parecer uma boa solução, ela faz com que expressões simples tenham tipos ambígüos. Como exemplo, considere: `n = x + y`, onde `x` e `y` são do tipo `Int`. Como existem duas possíveis instâncias com o tipo `Int` para as variáveis `a` e `b`, não é possível para o compilador decidir qual o tipo para `n` (este poderia ser `Int` ou `Float`).

Uma solução para este problema é a utilização de *dependências funcionais* [Jon00]. A idéia é declarar explicitamente uma relação de dependência entre os parâmetros da classe. Utilizando dependências funcionais, a classe `Add` ficaria:

```
Add a b r | a b → r where ...
```

A expressão `a b → r` significa que a partir dos tipos `a` e `b` consegue-se identificar o tipo `r`. Dependências funcionais são utilizadas para restringir 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 que variáveis de tipo podem assumir em tipos polimórficos. Com a definição da dependência funcional `a b → r` na classe `Add`, não é possível definir simultaneamente as seguintes instâncias:

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

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

Dependências funcionais são uma extensão ao padrão Haskell 98, mas como pode ser observado pela leitura da lista de discussão de Haskell, ou por comentários de usuários do compilador GHC, dependências funcionais e classes de tipos com múltiplos parâmetros parecem ser indispensáveis para muitos projetos.

Apesar do sistema de classes tipo utilizado pela linguagem Haskell parecer uma solução elegante para o polimorfismo de sobrecarga, este apresenta alguns problemas e limitações que serão abordadas na próxima seção.

3.1.6 Problemas com o de Polimorfismo de Sobrecarga em Haskell

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

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

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

Maiores detalhes sobre problemas relacionados a abordagem de polimorfismo de sobrecarga de Haskell podem ser encontrados em [CF99b, CaLFN07].

3.2 Polimorfismo de Sobrecarga no Sistema CT

Haskell utiliza uma abordagem de sobrecarga de *mundo aberto*, exigindo que todo símbolo sobrecarregado tenha seu tipo anotado em uma classe de tipo.

O sistema *CT*[CF99b, CaLFN07] estende o sistema de tipos de Damas-Milner com suporte ao polimorfismo de sobrecarga, utilizando uma abordagem de *mundo fechado*[4]. O sistema CT, ao contrário da linguagem Haskell, não exige que o programador declare uma classe de tipo somente para sobrecarregar símbolos. Ao invés disso, o sistema *CT* permite que o programador defina operações sobrecarregadas a medida do necessário, utilizando *tipos com restrições*[5].

Um tipo com restrições é um tipo que possui um conjunto de restrições associadas às suas variáveis de tipo. Este conjunto limita os tipos para os quais as variáveis podem ser instanciadas, na forma de exigências sobre a existência de certos símbolos no contexto onde o tipo será instanciado.

Assim como Haskell [Jon93, Kae88], o sistema *CT* permite a sobrecarga de funções sobre diferentes construtores de tipos e com um número diferente de parâmetros.

[4]Uma abordagem de mundo fechado para polimorfismo de sobrecarga determina o tipo de uma expressão com base no contexto onde esta expressão ocorre. Em [CaLFN07] é apresentada uma definição formal de abordagens de mundo aberto e mundo fechado para sobrecarga.

[5]Constrained Types

Além de não haver a necessidade de anotar explicitamente o tipo de símbolos sobrecarregados, outras diferenças entre o sistema CT e o sistema de classes de tipo em Haskell são[CF99a]:

1. O sistema CT utiliza uma regra menos restritiva para a detecção de expressões ambígüas. Maiores detalhes sobre as regras de ambigüidade utilizadas pelo sistema CT serão dados na Seção 3.2.3.

2. O sistema CT permite que um maior conjunto de expressões sejam bem tipadas, uma vez que torna possível a definição de símbolos sobrecarregados que não podem ser traduzidos em declarações de instâncias da linguagem Haskell.

A próxima seção apresenta informalmente o sistema CT por meio de alguns exemplos. Definições formais deste sistema de tipos podem ser encontradas em [Vas04, CF99b, CF99a].

3.2.1 Sistema CT

No sistema CT, os tipos de expressões são tipos polimórficos com restrições. Um tipo polimórfico com restrições possui a seguinte forma: $\forall\alpha_1 \ldots \alpha_n.\kappa.\tau$, onde $\alpha_1 \ldots \alpha_n$, $n \geq 0$ é um conjunto de variáveis de tipos; κ é um conjunto (possivelmente vazio) de restrições que consistem de pares $o : \tau$, onde o é um símbolo sobrecarregado e τ é um tipo simples. Um contexto de tipos Γ é um conjunto finito de suposições de tipo $x : \sigma$, onde x é um símbolo e σ um tipo polimórfico com restrições. Um símbolo é dito ser sobrecarregado se este possuir mais de uma suposição de tipo em Γ.

O *tipo principal* de uma expressão e, em um contexto Γ, representa o conjunto de todos os tipos que podem ser derivados para e em Γ. O tipo principal de um símbolo sobrecarregado $\emptyset$ é obtido a partir da *generalização mínima*[6] do conjunto de suposições de tipos para $\emptyset$ em Γ.

Uma substituição de tipos é uma função finita de variáveis de tipo em tipos simples. Seja $S = \{\alpha_i \mapsto \tau_i\}^{i=1\ldots n}$ uma substituição e σ um tipo, $S\sigma$ representa o tipo obtido substituindo-se toda variável de tipo α, que ocorre livre em σ, por $S(\alpha)$.

Um tipo τ é dito ser a generalização mínima de um conjunto de tipos $\{\tau_1, \ldots, \tau_n\}$ se as seguintes condições são satisfeitas por τ:

- Existe um conjunto de substituições S_i, com $i = \{1, \ldots, n\}$, tal que $S_i\tau = \tau_i$
- Se existir um conjunto de substituições $S_i^{'}$, com $i = \{1, \ldots, n\}$ e um tipo $\tau^{'}$ tal que $S_i^{'}\tau^{'} = \tau_i$ então existe também uma substituição S tal que $S\tau^{'} = \tau$.

[6] *lcg - Least Common Generalization.*

A primeira condição expressa que τ é uma generalização do conjunto $\{\tau_1, \ldots, \tau_n\}$, e a segunda garante que τ é a generalização *mínima*.

Como exemplo, considere a função `insert` que insere um elemento em uma lista ordenada. Suponha que existam no contexto de tipos Γ as seguintes definições:

```
(==) ::  Int → Int → Bool
(==) ::  Char → Char → Bool
(<) ::  Int → Int → Bool
(<) ::  Char → Char → Bool
```

A definição da função `insert` em uma linguagem similar a Haskel, baseada no sistema CT poderia ser feita do seguinte modo:

```
insert x [] = [x]
insert x (y:ys) = if x == y then y:ys else y:(insert x ys)
```

Na inferência de tipo da subexpressão `x == y`, é constatado que `(==)` é um símbolo sobrecarregado. Seu tipo principal é obtido a partir da generalização mínima entre os tipos de suas definições em Γ, que neste caso é igual a $\alpha \to \alpha \to \mathit{Bool}$. Portanto, no contexto Γ, onde este símbolo está sendo usado, o tipo inferido para `(==)`:

$\{\texttt{(==)} : \alpha \to \alpha \to \mathit{Bool}\} \cdot \alpha \to \alpha \to \mathit{Bool}$

Essa restrição é adicionada ao tipo inferido para a definição da função `insert`, uma vez que o símbolo sobrecarregado é usado em sua definição. O novo contexto de tipos Γ' tem, portanto, com as seguintes suposições:

```
(==) ::  Int → Int → Bool
(==) ::  Char → Char → Bool
(<) ::  Int → Int → Bool
(<) ::  Char → Char → Bool
insert :  ∀α · {(==) : α → α → Bool} · α → [α] → [α]
```

Considere agora, a seguinte definição de `insert` para um tipo de dados que representa uma árvore binária:

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


insert x Leaf = Node x Leaf Leaf
insert x (Node y l r)
                  | x == y = Node y l r
                  | x < y = Node y (insert x l) r
                  | otherwise = Node y l (insert x r)
```

O tipo inferido para esta nova definição de `insert` é:

$$\forall\alpha \cdot \{(==) : \alpha \to \alpha \to Bool, (<) : \alpha \to \alpha \to Bool\} \cdot \alpha \to \text{Tree } \alpha \to \text{Tree } \alpha$$

O tipo inferido para `insert`, no contexto Γ_{insert}, que contém todas as suposições de Γ' mais a suposição de tipo correspondente à definição de `insert` para árvores, é:

$$\forall\alpha\forall\beta \cdot \{\texttt{insert:} \quad \alpha \to \beta\ \alpha \to \beta\ \alpha\} \cdot \alpha \to \beta\ \alpha \to \beta\ \alpha$$

Cabe ressaltar que nenhuma informação sobre as restrições em relação aos símbolos `(==)` e `(<)` foram adicionadas às restrições da função `insert` neste contexto. Quando uma das funções sobrecarregadas for selecionada, com base nos tipos requeridos no contexto onde a função é usada, a satisfazibilidade das restrições presentes é verificada. Como exemplo desta verificação, considere a expressão `insert 2 [1]`. Primeiramente, é realizada a verificação se a restrição $\{\texttt{insert:} \quad \alpha \to \beta\ \alpha \to \beta\ \alpha\}$ é satisfazível para listas de inteiros. Como existe em Γ uma suposição que satisfaz essa restrição, verificam-se as condições impostas nesta suposição. A restrição presente na definição de `insert` para listas de inteiros é $\{\texttt{(==):} \quad Int \to Int \to Bool\}$, que também é satisfeita em Γ.

Ao se definir um novo símbolo sobrecarregado, o tipo deste não tem que ser uma instância da generalização mínima das definições anteriores. Uma nova definição pode ocasionar a mudança da generalização mínima para um tipo mais geral. Ainda considerando os exemplos anteriores para a definição de `insert`, podemos introduzir uma nova definição, como a seguinte:

```
insert f x [] = [x]
insert f x (y:ys) = if f x y then y:ys else y : (insert f x ys)
```

O tipo principal para esta definição é $\forall\alpha.(\alpha \to \alpha \to Bool) \to \alpha \to [\alpha] \to [\alpha]$. Calculando a generalização mínima para todas as definições de `insert`, obtemos:

$$\forall\alpha \cdot \forall\beta \cdot \forall\gamma \cdot \{\texttt{insert:} \quad \alpha \to \beta \to \gamma\} \cdot \alpha \to \beta \to \gamma$$

A função *lcg*, que calcula a generalização mínima, é apresentada na Figura 3.8. Evidentemente esta é uma simplificação, uma vez que, *lcg* é uma relação. Para realizar esta simplificação escolhe-se um representante da classe de equivalência entre tipos que são generalizações mínimas de um dado conjunto de tipos[Vas04].

3.2.2 Política de Sobrecarga

Nem todos os conjuntos formados por pares $x : \sigma$, onde x é um símbolo e σ o tipo deste símbolo, formam contextos de tipos válidos, mas somente conjuntos que satisfazem a política de sobrecarga do sistema CT, que diz que suposições de tipos distintas de um

$$lcg(\mathbb{T}) = \tau \texttt{ onde } (\tau, \mathcal{S}) = lcg'(\mathbb{T}, \emptyset), \texttt{ para algum } S$$

$$lcg'(\{\tau\}, S) = (\tau, S)$$

$$\begin{array}{l} lcg'(\{C\tau_1 \ldots \tau_n, C'\tau'_1 \ldots \tau'_m\}, S) = \\ \quad \texttt{if } S(\alpha) = (C\tau_1 \ldots \tau_n, C'\tau'_1 \ldots \tau'_m) \texttt{ para algum } \alpha \texttt{ then } (\alpha, S) \\ \quad \texttt{else} \\ \qquad \texttt{if } n \neq m \texttt{ then } (\alpha', S \dagger \{\alpha' \mapsto (C\tau_1 \ldots \tau_n, C'\tau'_1 \ldots \tau'_m)\}) \\ \qquad\quad \texttt{onde } \alpha' \texttt{ é uma variável de tipo livre} \\ \qquad \texttt{else } C_0\tau''_1 \ldots \tau''_n \\ \qquad\quad \texttt{onde } (C_0, S_0) = \begin{cases} (C, S) & \texttt{if } C = C' \\ (\alpha, S \dagger \{\alpha \mapsto (C, C')\}) & \texttt{caso contrário,} \\ & \texttt{onde } \alpha \texttt{ é uma} \\ & \texttt{var. de tipo livre} \end{cases} \\ \qquad\qquad (\tau''_i, S_i) = lcg'(\{\tau_i, \tau'_i\}^{i=1..n}, S_{i-1}), \texttt{ para } i = 1, \ldots, n \end{array}$$

$$lcg'(\{\tau_1, \tau_2\} \cup \mathbb{T}, S) = lcg'(\{\tau, \tau'\}, S') \texttt{ onde } \begin{array}{ll} (\tau, S_0) & = lcg'(\{\tau_1, \tau_2\}, S) \\ (\tau', S') & = lcg'(\mathbb{T}, S_0) \end{array}$$

Figura 3.8: Generalização Mínima

mesmo símbolo em um contexto não podem se sobrepor, isto é, o sistema CT não permite a definição de instâncias sobrepostas. Tipos $\forall\overline{\alpha} \cdot \kappa \cdot \tau$ e $\forall\overline{\alpha}' \cdot \kappa' \cdot \tau'$, com $(\overline{\alpha} \cup \overline{\alpha}') \cap (\text{tv}(\tau) \cap \text{tv}(\tau')) = \emptyset$ se sobrepõem se τ e τ' são unificáveis, ou seja, existe uma substituição S tal que $S\tau = S\tau'$.

Em [Vas04] é especificada a definição da política de sobrecarga utilizada no sistema CT, como apresentada acima. Na Seção 5.1, apresentamos uma extensão a essa política de sobrecarga, para permitir a definição de símbolos com definições sobrecarregadas sobrepostas, ou seja, cujos tipos podem ser unificados.

3.2.3 Ambigüidade

Uma expressão e é dita *ambígua* se podem ser obtidas denotações distintas para ela, usando uma semântica definida indutivamente sobre a derivação de um tipo para e [Mit96].

O processo de verificação da satisfazibilidade das restrições de um tipo polimórfico $\sigma = \forall\overline{\alpha}.\kappa.\tau$, onde $\kappa = \{o_i : \tau_i\}^{i=1\ldots n}$, em um contexto de tipos Γ, envolve encontrar todas as suposições de tipo de o_i, em Γ, que satisfazem ao conjunto de restrições $\{o_i : \tau_i\}^{i=1\ldots n}$. Seja $\mathcal{S} = \{R_1, \ldots, R_n\}$ um conjunto de substituições R_j, $j \in \{1, \ldots, m\}$,

tal que $R_j(\{o_i : \tau_i\}^{i=1,\ldots,n}.\tau)$ é uma instância de σ que satisfaz o conjunto de restrições $\{o_i : \tau_i\}^{i=1\ldots n}$ em Γ.

O tipo σ é considerado ambíguo, no sistema CT, se e somente se:

$$\exists i, j \in \{1, \ldots, m\} \text{ tais que } i \neq j,\ R_i \neq R_j \text{ e } R_i\tau = R_j\tau$$

Isto é, se existirem duas ou mais substituições distintas que satisfazem ao conjunto de restrições $\{o_i : \tau_i\}^{i=1\ldots n}$ e que, aplicadas ao tipo τ, produzem a mesma instância de σ, então este tipo σ é ambíguo.

De acordo com esta definição de ambigüidade, a expressão $f\ o$ seria considerada ambígua em um contexto {`f : Int → Int, f : Int → Float, f : Float → Float, o : Int, o : Float`}, já que existiriam duas possíveis interpretações para $f\ o$ com tipo `Float`. Note, entretanto, que essa expressão teria uma única interpretação se usada com tipo `Int`. Portanto, pode ser desejável relaxar a definição de ambiguïdade, de modo a permitir que $f\ o$ possa ser usada com tipo `Int`. Uma nova definição de ambigüidade, que permitiria que um maior número de expressões fossem consideradas bem tipadas, seria definida da seguinte maneira. Sendo $\mathcal{S} = \{R_1, \ldots, R_n\}$ tal como anteriormente, o tipo σ seria considerado ambíguo somente se não existir um R_i $1 \leq i \leq n$ tal que $R_i\tau \neq R_j\tau$ para todo $1 < j < m$, $j \neq i$. Ou seja, a expressão somente é considerada ambígua quando usada em um contexto onde não existe a possibilidade de tradução única. É importante ressaltar que essa nova definição de ambigüidade contraria a definição usual de coerência, sendo necessária uma análise mais detalhada para determinar as vantagens e desvantagens de sua adoção.

3.2.4 Exemplos de Polimorfismo de Sobrecarga no Sistema CT

Em [Jon93] é apresentada uma alternativa para definições sobrecarregadas de símbolos sobre classes parametrizadas por construtores de tipos.

Uma linguagem baseada no sistema CT permite a definição de símbolos sobrecarregados sobre diferentes contrutores de tipos, como em Haskell, sem a necessidade de definir uma classe para anotar o tipo deste símbolo, uma vez que o sistema CT adota a abordagem de mundo fechado.

Como um primeiro exemplo, considere a função `map` definida nas bibliotecas padrão da linguagem Haskell como:

```
map :: (a → b) → [a]  → [b]
map f [] = []
map f (x:xs) = (f x) : (map f xs)
```

$$\forall\alpha.\forall\beta.\forall\gamma.\{map : (\alpha \to \beta) \to \gamma\alpha \to \gamma\beta\}.(\alpha \to \beta) \to \gamma\alpha \to \gamma\beta$$

$$\alpha \to \beta = lcg(\alpha \to \beta, \alpha \to \beta)$$
$$\gamma\ \alpha = lcg(Tree\ \alpha, Maybe\ \alpha)$$
$$\gamma\ \beta = lcg(Tree\ \beta, Maybe\ \beta)$$

Figura 3.9: Inferência do tipo da função `map` a partir da generalização minima de suas definições

Muitos programas funcionais utilizam funções similares a esta, para aplicar uma determinada função a todos os elementos de uma estrutura de dados. Em uma linguagem similar a Haskell, que utilize o sistema de tipos CT, poderíamos dar diversas definições de `map`, para diferentes tipos de dados, sem a necessidade de anotações de tipo. Como exemplo, considere as seguintes definições:

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
data Maybe a = Nothing | Just a
overload map _ Leaf = Leaf
         map f (Node a l r) = Node (f a) (map f l) (map f r)

overload map _ Nothing = Nothing
         map f (Just x) = Just (f x)
```

A primeira definição, aplica uma função a todos os elementos de uma árvore binária. O tipo inferido para esta função é `map : (a → b) → Tree a → Tree b`. A segunda definição de `map`, sobrecarregada sobre o tipo `Maybe`, possui o tipo `map : (a → b) → Maybe a → Maybe b`. A Figura 3.9 mostra resumidamente o processo de obtenção do tipo de `map` em um contexto que contenha essas definições.

Note, que esta mesma funcionalidade poderia ser obtida em Haskell pelo seguinte trecho de código:

```
class Functor m where
   map ::  (a→b)→m a→m b

instance Functor [ ] where
   -- map ::  (a→b)→[a]→[b]
   map f [ ] = [ ]
   map f (x:xs) = (f x) : (map f xs)

instance Functor Tree where
```

```
   -- map ::  (a→b)→Tree a→Tree b
   map f Leaf = Leaf
   map f (Node a l r) = Node (f x) (map f l) (map f r)

instance Functor Maybe where
   -- map ::  (a→b)→Maybe a→Maybe b
   map f Nothing = Nothing
   map f (Just x) = Just (f x)
```

Outro tipo de função comumente utilizada em programas funcionais é a contagem do número de elementos em uma estrutura de dados. Na biblioteca padrão de Haskell, existe a função `length` que conta o número de elementos presentes em uma lista. Considere a definição de `length` para o tipo de dados `GRose`:

```
data GRose f a = GNode a (f (GRose a))
```

Esta estrutura de dados representa uma árvore n-ária parametrizada por outra estrutura para armazenagem dos descendentes de um nodo. Na definição deste tipo, a variável `f` representa um construtor de tipos de kind $\star \rightarrow \star$[7]. Como um exemplo de valor deste tipo, considere a seguinte constante:

```
t = (GNode 1 [(GNode 2 ([GNode 3 []])), (GNode 4 ([]))])
```

O tipo de `t` é `GRose [] Int`, onde o contrutor de tipos `f` é instanciado como o tipo de listas.

A seguir é mostrada a definição de `length` para o tipo `GRose`:

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
data GRose f a = GNode a (f (GRose a))
overload length :: Int → Int
         length _ = 1

overload length Leaf = 0
         length (Node _ l r) = 1 + (length l) + (length r)

overload length [] = 0
         length (x:xs) = (length x) + (length xs)

overload length (GNode x xs) = 1 + length xs
```

[7] *Kinds* classificam tipos da mesma maneira que tipos classificam valores. O kind $\star$ é dito ser o kind de "tipos", portanto, se `f` possui o kind $\star \rightarrow \star$, então `m` mapeia um tipo em outro.

O tipo inferido para `length` em um contexto contendo os tipos das definições acima é:

```
length::  a → Int
```

Que corresponde a generalização mínima de:

```
length ::  {length ::  a → Int} ⇒ [a] → Int

length ::  Tree a → Int

length ::  Int → Int

length ::  {length ::  f (GRose f a) → Int }⇒ GRose f a→ Int
```

Com isto, tem-se que a expressão `y = length t` é bem tipada neste contexto, uma vez que, como `t` é uma constante de tipo `GRose [] Int`, temos que a restrição associada ao símbolo `length` é satisfeita neste contexto, pela implementação deste símbolo para listas e para números inteiros.

Observe que, no trecho de código anterior, para definirmos `length` para o tipo `Int`, foi necessário acrescentar uma anotação de tipos. Caso esta fosse omitida, seria inferido o tipo `length ::  a → Int`, que não seria aceito pela política de sobrecarga do sistema CT por esta instância de `length` se sobrepor às outras definições deste símbolo no mesmo contexto. No Capítulo 5, será apresentada uma extensão à política de sobrecarga do sistema CT para permitir definições de instâncias sobrepostas.

3.3 Conclusão

Neste capítulo foram apresentados duas abordagens para o polimorfismo de sobrecarga. A primeira é utilizada pela linguagem Haskell e a segunda é utilizada no sistema CT. Para cada uma destas abordagens foram apresentadas suas características principais e exemplos. O polimorfismo de sobrecarga é um dos ingredientes necessários para a definição de funções genéricas utilizando representação estrutural de tipos algébricos. O próximo capítulo apresentará abordagens para programação genérica em Haskell.

Capítulo 4

Programação Genérica

A tarefa de desenvolver programas muitas vezes consiste em projetar um tipo de dados, e adicionar funcionalidades ao mesmo [HJL06]. Algumas destas funcionalidades são definidas para todos os tipos, independentemente da natureza destes, e outras possuem um comportamento similar, mas que varia para cada tipo em que a funcionalidade é utilizada. As primeiras podem ser facilmente implementadas em linguagens com suporte a *polimorfismo paramétrico* (Seção 1.1). As do segundo tipo são conhecidas na literatura[HJL06, Löh04] como *funcionalidades genéricas* e podem ser implementadas em linguagens com suporte a *polimorfismo de sobrecarga* ou em linguagens com suporte a *programação genérica* (*polimorfismo estrutural*). Alguns exemplos de funcionalidades genéricas são: armazenar um valor em um banco de dados, comparar dois valores, converter valores para *String* (*pretty-print*), *parsing* etc. A lista não se limita a estes pequenos exemplos, incluindo também:

- Ferramentas para manipulação de XML, como *compressores*[HJ02a], e ferramentas para *data-binding*;
- Ferramentas para construção automática de testes[KATP02];
- Código para percorrer tipos de dados (*boilerplate code*)[LJ04, LJ03].

A necessidade de realizar mudanças, durante o ciclo de vida de um programa, é muito freqüente. Regras de negócio e tecnologia mudam freqüentemente, levando à necessidade de modificações no programa. Todavia, alterar um grande sistema de software é uma tarefa desafiadora: localizar o código que é responsável por uma funcionalidade, alterá-lo e garantir que esta mudança não afete outros componentes é muito difícil[Pow].

Um importante tipo de mudança é a alteração dos tipos de dados utilizados pelo programa. Mudanças na representação de informação acontecem, na grande maioria

das vezes, devido a alteração de requisitos do software. Se um tipo de dados muda ou um novo tipo de dados é adicionado, o programa deve ser adaptado para que fique coerente com a mudança realizada. Um programa genérico adequa-se automaticamente a esta mudança, sem a necessidade de intervenção do programador [Löh04, HJL06]. Por isto, pesquisadores acreditam que a programação genérica possui potencial para resolver uma importante parte do problema de *evolução de software*[JP06].

O restante deste capítulo visa apresentar algumas propostas para programação genérica em Haskell, por meio de alguns exemplos canônicos. Na proxima seção, são apresentadas, de maneira sucinta, duas abordagens: *Derivable type classes*[HJ00] e *Generics for the masses* [Hin04, dSOHL]. Outras duas abordagens serão apresentadas com mais detalhes: *Scrap your boilerplate*[LJ03, LJ04, LP05], que permite definir programas genéricos em termos de operadores de caminhamento sobre tipos de dados, e *Generic Haskell*[LCJ03, Löh04, HJ02b, HJ02a], que consiste de uma extensão de Haskell para programação genérica. Finalizamos este capítulo apresentando uma análise comparativa entre *Generic Haskell* e *Scrap your boilerplate*, para isto definindo alguns critérios para direcionar essa análise.

4.1 Programação Genérica em Haskell

4.1.1 Representação Estrutural de Tipos de Dados

Diversas abordagens de programação genérica em Haskell fazem uso de uma representação algébrica de tipos de dados. Esta seção visa introduzir esta representação. Todo tipo de dado em Haskell pode ser visto como uma soma de produtos[Löh04], o que equivale a dizer que a representação estrutural de qualquer tipo de dado Haskell pode ser feita utilizando os seguintes tipos:

```
data Unit = Unit
data Sum a b = Inl a | Inr b
data Prod a b = Prod a b
data Con a = Con a
data Label a = Label a
```

O conjunto de construtores de dados de um tipo é representado como uma soma utilizando o tipo `Sum` (que tem associatividade à direita). Produtos, com $n \geq 2$ parâmetros, são representados utilizando o tipo `Prod` (que também tem associatividade à direita), produtos constituídos de um único parâmetro são construídos pelo próprio tipo e produtos sem parâmetros são representados pelo tipo `Unit`. O tipo `Con`

representa o rótulo de um construtor de dados e o tipo `Label` um rótulo de registro. Um tipo registro pode ser representado naturalmente como um produto com rótulos.

Como exemplo considere o seguinte tipo de dados:

```
data List a = Nil | Cons a (List a)
```

Este tipo é composto pela soma de dois produtos: `Nil` e `Cons a (List a)`. O primeiro produto corresponde a um produto nulo, sendo, portanto, representado pelo tipo `Unit`. Por sua vez, o segundo tipo consiste no produto de um elemento do tipo `a` por um elemento de tipo `List a`. Sendo assim, a representação estrutural deste tipo é:

```
type ListRep a = Sum (Con Unit) (Con (Prod a (List a)))
```

Tipos como `ListRep` são chamados *tipos de representação estrutural*. Um tipo `t` e sua representação estrutural t' são isomórficos. Este isomorfismo pode ser representado por um par de funções de tipo $a \rightarrow b$ e $b \rightarrow a$, que podemos incluir em um tipo de dados, como a seguir:

```
data EP a b = EP (a → b) (b → a)
```

Por exemplo, o valor do tipo `EP` que contém as funções de conversão do tipo `List` seria: $conv_{List}$ `= EP` $(from_{List})$ (to_{List}), onde $(from_{List})$ e (to_{List}) são definidas por:

$(from_{List})$ `::` $\forall a.$ `List a` $\rightarrow$ `List'a`
$(from_{List})$ `Nil = Inl (Con Unit)`
$(from_{List})$ `(Cons a as) = Inr (Con (Prod a as))`

(to_{List}) `::` $\forall a.$ `List'a` $\rightarrow$ `List a`
(to_{List}) `(Inl (Con Unit)) = Nil`
(to_{List}) `(Inr (Con (Prod a as))) = Cons a as`

Figura 4.1: Convertendo valores de listas para sua representação estrutural e vice-versa.

A partir destas representações, um compilador pode gerar definições específicas de uma função definida sobre a estrutura de um tipo de dados. Esta abordagem para representação estrutural de tipos é utilizada pela linguagem *Generic Haskell*.

4.1.2 Derivable Type Classes

Um das características mais inovadoras de Haskell é a sua abordagem para o polimorfismo de sobrecarga baseada em classes de tipos. Classes de tipo podem, opcionalmente, fornecer implementações padrão (*default*) para seus símbolos sobrecarregados. Baseado

neste ponto de vista, Hinze propôs uma extensão à linguagem Haskell que permite a definição de métodos *default* genéricos, que recebeu o nome de *Derivable Type Classes*.

A idéia principal desta abordagem é permitir ao programador definir métodos padrão, em classes de tipos, utilizando indução estrutural sobre a representação algébrica do tipo de dados.

Como um exemplo, (retirado de [HJ00]), considere a seguinte implementação genérica da classe `Eq`:

```
class Eq a where
    (==), (/=) :: a → a → Bool

    (==) {Unit} Unit Unit = True
    (==) {Sum a b} (Inl x) (Inl y) = x == y
    (==) {Sum a b} (Inr x) (Inr y) = x == y
    (==) {Sum a b} _ _ = False
    (==) {Prod a b} (Prod x y) (Prod z w) = (x == z) && (y == w)


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

A partir desta definição genérica, um compilador Haskell pode gerar automaticamente definições de `(==)` e `/=` para cada instância da classe `Eq`. Como exemplo, pode-se definir uma instância da classe `Eq` para árvores binárias simplesmente escrevendo:

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
instance (Eq a) ⇒ Eq (Tree a)
```

Todo o código específico para a instância da classe `Eq` para o tipo algébrico `Tree` é gerado pelo compilador. Observe que utilização desta abordagem pode ser vista como uma alternativa para o mecanismo de derivação automática de instâncias presente para algumas classes de tipos de Haskell (especificando, na definição do tipo algébrico, uma cláusula *deriving*).

No artigo original desta abordagem[HJ00] são apresentados alguns de seus pontos positivos: classes de tipos podem especificar tanto métodos *default* genéricos e não genéricos. Tanto métodos genéricos e não genéricos podem ser mutamente recursivos. Além disso, declarações de funções genéricas somente aparecem em declarações de classes. Apesar desta proposta proporcionar uma maneira simples para a definição de funções genéricas, as seguintes desvantagens podem ser apontadas: declarações genéricas só podem ser dadas para instâncias de classes de tipos que tenham variáveis de tipo de kind $\star$. Esta abordagem não permite definir métodos genéricos para classes de tipos com múltiplos parâmetros.

4.1.3 Generic for the Masses

Como diversas propostas para programação genérica em Haskell demandavam algum tipo de extensão à linguagem, Hinze propôs, em [Hin04], uma abordagem para o desenvolvimento de funções genéricas, de nome *Generic for the Masses*, que pode ser totalmente expressa utilizando o padrão Haskell 98.

A idéia desta abordagem é utilizar uma representação estrutural de tipos de dados, juntamente com algumas operações sobrecarregadas em classes de tipos. Como a representação estrutural utilizada por esta abordagem é similar à apresentada na Seção 4.1.1, esta será omitida. Leitores interessados podem consultar [Hin04].

Para a implementação de funções genéricas são utilizadas duas ou três classes de tipos, dependendo do tipo de função genérica que se deseja definir:

- **Generic**: Esta classe define operações que serão aplicadas sobre a estrutura algébrica de tipos de dados.

- **TypeRep** : Esta classe define uma operação que retorna a representação estrutural do tipo para o qual uma instância desta classe foi definida. A representação estrutural é definida em termos das operações definidas na classe `Generic`.

- **FunctorRep**: Usada para permitir a definição de funções genéricas sobre diferentes construtores de tipos. Deve-se utilizar esta terceira classe quando é necessário realizar a conversão de um construtor de tipo em sua correspondente estrutura algébrica. Assim como a classe TypeRep, a classe FunctorRep também utiliza as operações definidas em Generic para construir a representação do tipo para o qual uma instância de FunctorRep foi definida.

A seguir é mostrado o código das classes `Generic` e `TypeRep` que permitem definir funções genéricas:

```
class Generic g where
    unit :: g Unit
    plus :: (TypeRep a, TypeRep b) ⇒ g (Plus a b)
    pair :: (TypeRep a, TypeRep b) ⇒ g (Pair a b)
    datatype :: (TypeRep a) ⇒ Isomorphism a b → g b
    char :: g Char
    int :: g Int
class TypeRep a where
    typeRep :: (Generic g) ⇒ g a
```

Observe que a classe `Generic` possui símbolos para representar cada tipo primitivo (`Char`, `Int`) e tipos que representam a estrutura algébrica de definições de tipos em Haskell (`plus` para somas, `pair` para produtos binários e `unit` para valores cujos construtores não possuem parâmetros). O símbolo `datatype` é responsável por obter representações de valores de tipos de dados algébricos em geral, este recebe como parâmetro um valor do tipo `Isomorphism a b`, mostrado a seguir:

```
data Isomorphism a b = Isomorphism {fromData :: b → a, toData :: a → b}
```

Este é composto por um par de funções que são responsáveis, respectivamente, por converter valores de um tipo algébrico qualquer para uma outra representação e desta para valores do tipo algébrico.

Para um exemplo concreto, considere o problema de converter valores de um tipo algébrico qualquer em um string de bits. Primeiramente deve-se definir tipos para representar cadeias de bits:

```
data Bit = Zero | One deriving Show
type Bin = [Bit]
```

Uma função que converte valores de qualquer tipo para uma cadeia de bits deve ter o tipo `a →[Bit]`. Para definir esta assinatura de função genérica, [Hin04] utilizou uma nova declaração de tipo:

```
newtype ShowBin a = ShowBin{appShowBin ::  a → Bin}
```

A conversão de um valor de um tipo algébrico qualquer em uma cadeia de bits só é possível se este for expresso via sua representação estrutural. Para isso, deve-se criar uma instância da classe `TypeRep`, que realiza a conversão de um valor de um tipo qualquer em sua respectiva representação. Como exemplo, considere as seguintes instâncias de `TypeRep` para listas e números inteiros:

```
instance TypeRep Int where
   typeRep = int
instance (TypeRep a) ⇒ TypeRep [a] where
   typeRep = datatype (Isomorphism {fromData = fromList, toData = toList})
```

As funções `fromList` e `toList` possuem implementações idênticas às apresentadas na Figura 4.1.

Tendo sido esta instância de `TypeRep` definida, o programa genérico para realizar a conversão é dado por uma função e uma declaração de instância, da classe `Generic`, que são mostradas a seguir:

```
showBin :: (TypeRep a) ⇒ a → Bin
showBin = appShowBin typeRep
```

Figura 4.2: Função genérica para conversão de um valor em uma cadeia de bits

Nesta definição, `typeRep` é o método sobrecarregado definido na classe `TypeRep` e `appShowBin` é uma função de projeção associada à definição do registro `ShowBin`. A definição acima não revela os detalhes associados a implementação do algoritmo que realiza a conversão em bits. Tais detalhes são fornecidos pela instância da classe `Generic` cujo código realiza a análise estrutural do tipo em questão. O código da instância desta classe é mostrado abaixo:

```
instance Generic ShowBin where
   unit = ShowBin (λx →[])
   plus = ShowBin (λx → case x of
                              Inl l → Zero : showBin l
                              Inr r → One : showBin r
   pair = ShowBin(λx→ showBin(outl x)++showBin(outr x))
   dataype iso = ShowBin (λx→showBin (fromData iso x))
   char = ShowBin (λx → bits 7 x)
   int = ShowBin (λx → bits 16 x)
```

Veja que esta instância define o algoritmo de conversão como uma análise de casos sobre a estrutura de um tipo arbitrário. A relação entre a função `showBin` e esta instância de `Generic` é realizada pela função sobrecarregada `typeRep`, cuja implementação para listas foi mostrada anteriormente.

Apesar desta abordagem possuir a vantagem de ser totalmente representável dentro do padrão Haskell 98 e conseguir definir funções genéricas sobre construtores de tipos (utilizando a classe `FunctorRep`), ela possui o inconveniente de não ser expressiva o bastante para definir funções genéricas que são parametrizadas por mais de um parâmetro de tipo[Hin04]. Para implementar funções genéricas que são parametrizadas por dois ou mais argumentos de tipo é necessária a criação de uma classe de tipos diferente para cada aridade de parâmetros envolvidos.

4.2 Scrap Your Boilerplate

Programadores Haskell lidam freqüentemente com tipos de dados complexos e necessitam, muitas vezes, aplicar funções a trechos específicos que podem estar arbitrariamente aninhados em valores destes tipos. Estas aplicações requerem o uso do que é chamado

de *boilerplate code*, que realiza a travessia de um tipo recursivamente, possivelmente reconstruindo valores e aplicando funções aos termos desejados. Este tipo de código é de difícil manutenção, uma vez que, se a estrutura do tipo em questão muda, o código deverá ser alterado. Para tipos de dados com um número reduzido de construtores de dados, realizar a alteração deste código não é problema. Mas se este código manipula tipos de dados contendo muitos construtores ou um conjunto de tipos de dados mutuamente recursivos, a tarefa de alterar tal código se torna invariavelmente complexa. O padrão de projeto *Scrap Your Boilerplate (SYB)*[LJ04, LJ03, LP05] tenta resolver este problema, definindo uma biblioteca de combinadores para construir caminhamentos genéricos sobre tipos de dados Haskell. Com esta abordagem, programadores concentram-se apenas em escrever o código para implementação da funcionalidade desejada, utilizando combinadores para expressar como a estrutura de dados será percorrida.

Funções genéricas em *SYB* são aplicáveis a todos os tipos da classe `Data`. Esta classe provê operações, explicadas a seguir, para desconstruir ou construir valores de um tipo, além de prover métodos para obter informações sobre a estrutura deste tipo. Observe que a classe `Data` é uma sub-classe de `Typeable`, que será apresentada e explicada a seguir. Primeiramente, consideraremos a classe `Data` e seus respectivos símbolos.

```
class (Typeable a) ⇒ Data a where
    toConstr ::  a → Constr
    dataTypeOf ::  a → DataType
    gfoldl ::  ∀ a f.(∀a b.Data a ⇒ f (a→b)→a→f b)→
              (∀a.a→f a)→a→ f a
```

Figura 4.3: Trecho da definição da classe `Data`

A função `toConstr` produz informação sobre o construtor de dados que construiu um dado valor. O tipo de dados `Constr` é abstrato e pode ser utilizado para a obtenção de valores como o nome do construtor ou a que tipo de dados este pertence.

De maneira análoga, `dataTypeOf` retorna um valor do tipo abstrato `DataType` que representa o tipo de dados do valor fornecido como parâmetro.

A função `gfoldl` permite que valores de um tipo `a` qualquer (terceiro argumento) seja descontruído em um resultado de tipo `f a`. Quase todo valor representável em Haskell é uma aplicação de um construtor de dados a outros valores (cada valor de um tipo primitivo pode ser visto como um construtor sem parâmetros). Se um valor v possui a seguinte forma(onde C é um construtor de dados e v_i, $1 \leq i \leq n$ são argumentos deste construtor):

$$C\ v_1\ v_2 \ldots\ v_n$$

então `gfoldl` $\diamond$ c v é igual a:

$$(\ldots((c\ C \diamond v_1) \diamond v_2) \diamond \ldots \diamond v_n).$$

O segundo argumento, c, é aplicado à função representada pelo construtor de dados C e cada aplicação, presente no valor $v = C\ v_1\ v_2 \ldots\ v_n$, é alterada, inserindo-se o primeiro argumento ($\diamond$) entre os componentes envolvidos nesta aplicação. Para um exemplo concreto da implementação desta função, considere os seguintes tipos de dados:

```
data T a b = C1 a b | C2
data [a] = a : [a] | []
```

Abaixo são mostradas possíveis implementações de `gfoldl` para estes tipos:

```
instance (Data a, Data b) ⇒ Data (T a b) where
  gfoldl k z (C1 a b) = z C1 `k` a `k` b
  gfoldl k z C2 = z C2
  ...
instance Data a ⇒ Data [a] where
  gfoldl k z [] = z []
  gfoldl k z (x : xs) = z (:) `k` x `k` xs
  ...
```

Todos os combinadores fornecidos pelo *SYB* são construídos utilizando o conjunto de operações primitivas para a obtenção de informações sobre tipos, para processar aplicações de construtores (fornecidas pela classe `Data`) e para realizar uma conversão de tipo de maneira segura. Esta última operação é definida pela classe `Typeable`, cujo código é apresentado e explicado a seguir:

```
class Typeable a where
   typeOf :: a → TypeRep
data TypeRep = TR String [TypeRep]
instance Typeable Int where
   typeOf x = TR "Prelude.Int" []
instance Typeable a ⇒  Typeable [a] where
   typeOf x = TR "Prelude.List" [typeOf (get x)]
              where
                 get :: [a] → a
                 get = undefined
instance (Typeable a, Typeable b) ⇒ Typeable(a→b) where
   typeOf f = TR "Prelude.->"[typeOf (getArg f), typeOf getRes f]
```

```
        where
            getArg ::(a→b)→a
            getArg = undefined
            getRes ::(a→b)→b
            getRes = undefined
```

Observe que esta classe possui um único método, `typeOf`, que tem a função de gerar uma representação intermediária de um tipo de dados. Note que, em todas as instâncias apresentadas, `typeOf` *nunca avalia seu argumento.* A chamada `(get x)`, na instância para listas, não será avaliada; ela simplesmente instancia a variável de tipo na chamada recursiva de `typeOf`, para o tipo que corresponde ao tipo dos elementos da lista[LJ03]. Como exemplo de como ocorre a instanciação desta variável de tipo, considere a seguinte expressão: `g = typeOf [2]`. É evidente que o tipo de `[2]` é `[Int]`. Durante a avaliação de `g`, é realizada a chamada de `typeOf` que está presente na instância de `Typeable` para listas. Observando a equação correspondente:

```
typeOf x = TR "Prelude.List" [typeOf (get x)]
```

vemos que, durante a execução, a variável `x` é ligada ao valor `[2]` e este é passado para a chamada recursiva de `typeOf`, que recebe como argumento o valor retornado pela função `get`. Nesta função, o tipo de `get` será instanciado para `[Int]` → `Int`, que permite que a chamada de `typeOf` seja feita para a instância de `Typeable` correspondente ao tipo `Int`, fazendo com que tenhamos `g = TR "Prelude.List"[TR "Prelude.Int"[]]`, sem que haja a necessidade de avaliação do argumento `x`.

A partir da definição de `typeOf`, é possível implementar um operador de coerção de tipo cuja segurança depende de uma implementação correta de `typeOf`. O código para este é:

```
cast :: (Typeable a, Typeable b) ⇒ a → Maybe b
cast x = r
        where
            r = if typeOf x == typeOf (get r)
                then Just (unsafeCoerce x)
                else Nothing
            get :: Maybe a → a
            get x = undefined
```

Esta definição utiliza a extensão `unsafeCoerce` que possui o tipo `unsafeCoerce :: a → b`. De acordo com [LJ03] esta função apenas realiza uma mudança do tipo do valor fornecido como argumento, sem realizar uma mudança da representação do valor

```
mkT :: (Typeable a, Typeable b) ⇒ (b → b) → a → a
mkT f = case cast f of
            Just g  → g
            Nothing → id
```

Figura 4.4: Definição de mkT.

em si. Internamente ao compilador GHC, esta função não é implementada e também não é uma função primitiva; esta consiste apenas em um identificador que é criado internamente pelo próprio compilador, no módulo *MkId.lhs* (presente no código fonte do GHC).

Um operador de coerção de tipo pode ser utilizado para *estender* uma função que opera sobre um único tipo t para uma que opera sobre diversos tipos, mas comporta-se como a função identidade em todos os tipos exceto t. A função `mkT` que realiza esta extensão é mostrada na Figura 4.4 e explicada a seguir: `mkT f x` aplica `f` a `x`, se o tipo de `x` for igual ao argumento de `f`; caso contrário, a função identidade é aplicada a `x`. Para uma melhor compreensão, considere os seguintes exemplos, obtidos em uma sessão interativa no interpretador *GHCi*[1]:

```
Prelude> (mkT not) True
False
Prelude> (mkT toUpper) False
False
```

No primeiro exemplo, o parâmetro utilizado pela função `not` possui o mesmo tipo do valor `True`, o que faz com que o tipo de `mkT` seja instanciado para `(Bool → Bool) → Bool → Bool`. Durante a execução de `mkT`, a variável `f` denotará a função `not` que, em seguida, será passada como argumento para `cast`. Esta última função terá seu tipo instanciado em `(Bool → Bool) → Maybe Bool`, o que faz com que a implementação da função `typeOf` presente na instância de `Typeable` para o tipo `Bool` seja executada. Como os valores retornados por `typeOf` são iguais, a coerção é realizada e a função `not` é aplicada ao valor `True` produzindo o resultado `False`.

Por sua vez, no segundo exemplo, `mkT` realiza uma extensão da função `toUpper`. Como `toUpper` possui o tipo `Char→Char` e a função `(mkT toUpper)` é aplicada ao valor `False`, de tipo `Bool`, o tipo de `mkT` é instanciado em `(Char → Char) → Bool → Bool`. Durante a execução de `mkT`, a variável `f` denotará a função `toUpper`, que em seguida, será passada como argumento para `cast`. Esta última função terá seu tipo

[1]As funções `not` e `toUpper` estão definidas na biblioteca da linguagem Haskell e seus tipos são respectivamente `Bool→Bool` e `Char→Char`

instanciado em `(Char → Char) → Maybe Bool` (note que o tipo de retorno desta função é instanciado com o mesmo tipo de retorno de `mkT`), o que faz com que a implementação da função `typeOf` presente nas instâncias de `Typeable` para os tipos `Bool` e `Char` sejam executadas. Como os valores retornados por `typeOf` são diferentes, a coerção não é realizada e a função identidade é aplicada ao valor `False`, produzindo o resultado final.

O operador de coerção pode ser utilizado para a criação de uma função similar a `mkT` para a criação de consultas - a função `mkQ`. A consulta `(r ‘mkQ‘q)`, quando aplicada a um argumento de tipo `a`, aplica `q` ao valor de tipo `a` se `q` espera um valor deste tipo como argumento; caso contrário, um valor padrão `r` é retornado.

```
mkQ :: (Typeable a, Typeable b) ⇒ r → (b → r) → a → r
(r ‘mkQ‘ q) a = case cast a of
                    Just b  → q b
                    Nothing → r
```

Além das primitivas fornecidas pelas classes `Data` e `Typeable` já apresentadas, a classe `Data` fornece diversos combinadores que podem ser utilizados para construir estratégias de caminhamento mais específicas. Estes combinadores são `gmapT`, `gmapQ` e `gmapM`. O combinador `gmapT` recebe como parâmetro uma função polimórfica e um valor de tipo também polimórfico `a` e aplica esta função a todos descendentes imediatos deste valor, retornando como resultado um valor deste tipo `a`. Seja $v = C\ t_1 \ldots t_n$, então uma definição esquemática para esta função é:

$$\text{gmapT}\ f\ (C\ t_1 \ldots t_n) = C(f\ t_1) \ldots (f\ t_n)$$

Para valores que não possuem nodos *filhos*, `gmapT` comporta-se como a função identidade. Como exemplo, considere a seguinte definição presente na instância da classe `Data` para o tipo `Int`:

```
instance Data Int where
   gmapT _ x = x
```

Para tipos definidos pelo usuário, as definições de `gmapT` seguem o esquema anterior, como pode ser visto nos exemplos a seguir:

```
data T a b = C1 a b | C2 b
data Tree a = Leaf  | Node a (Tree a) (Tree a)
instance (Typeable a, Typeable b) ⇒ Data (T a b) where
   ...
   gmapT f (C1 a b) = C1 (f a) (f b)
```

```
   gmapT f (C2 b) = C2 (f b)
instance (Typeable a, Typeable b) ⇒ Data (Tree a) where
   ...
   gmapT f Leaf = Leaf
   gmapT f (Node a l r) = Node (f a) (f l) (f r)
```

Note que na instância da classe `Data` para árvores, a função `f` é aplicada ao valor de tipo `a` e às sub-árvores esquerda (variável l) e direita (variável r), ao invés de utilizar uma chamada recursiva para cada sub-árvore. Evidentemente, podemos definir `gmapT` utilizando recursão (assim como é feito para a função `map` definida para listas), aplicando `(gmapT f)` a cada uma destas sub-árvores conforme mostrado no trecho de código a seguir:

```
instance (Typeable a, Typeable b) ⇒ Data (Tree a) where
   ...
   gmapT f Leaf = Leaf
   gmapT f (Node a l r) = Node (f a) (gmapT f l) (gmapT f r)
```

Porém, definir `gmapT` como um operador de um nível (aplicando a função de parâmetro a todos os descendentes imediatos), permite que diferentes estratégias de caminhamento sejam definidas a partir deste. Como por exemplo, considere o problema de desenvolver combinadores que apliquem uma determinada função `f` em todos os componentes de uma estrutura em caminhamentos *bottom-up* e *top-down*:

```
bottomUp f = f .  gmapT (bottomUp f)

topDown f x = gmapT (topDown f) (f x)
```

A função `bottomUp` primeiramente, aplica `(bottomUp f)` em todos os descendentes imediatos de um determinado valor, e ao final aplica a função `f` ao resultado desta chamada recursiva. Como a aplicação recursiva de `(bottomUp f)` é realizada por meio do combinador `gmapT`, a função `f`, é aplicada a cada componente do valor em um caminhamento *bottom-up*, nível a nível. Por sua vez, `topDown` aplica `f` ao valor `x`, e em seguida, aplica `topDown f` a todos os nodos filhos do valor resultante da aplicação `f x`, resultando na aplicação de `f` a cada componente do valor `x` em um caminhamento *top-down*.

Sabe-se que é possível expressar uma função `map` para um tipo algébrico qualquer em termos de seu *catamorfismo*[MFP91]. Como `gfoldl` é um *catamorfismo* sobre a estrutura de aplicação de contrutores de dados e a função `gmapT` aplica uma função fornecida como parâmetro a cada um dos componentes envolvidos na estrutura de

aplicação que forma um valor de um tipo algébrico qualquer, pode-se definir `gmapT` utilizando `gfoldl`. Para entender como `gmapT` pode ser expresso via `gfoldl`, considere a implementação destas duas funções para um mesmo tipo de dados, apresentadas a seguir:

```
data T a b = C1 a b
instance (Typeable a, Typeable b) ⇒ Data (T a b) where
   gmapT f    (C1 a b) = C1 (f a) (f b)
   gfoldl k z (C1 a b) = (z C1) `k` a `k` b
   ...
```

Observando ambas as definições, pode-se perceber que o construtor `C1`, em `gmapT`, não é parâmetro de nenhuma função. Portanto, o parâmetro `z` de `gfoldl` deve ser instanciado como a função identidade. Por sua vez, o parâmetro `k` de `gfoldl` pode ser definido para aplicar `f` ao seu segundo argumento e então aplicar o seu primeiro argumento, ou seja, `gmapT` pode ser expressa em termos de `gfoldl` como:

```
gmapT :: (∀ b . Data b ⇒ b → b) → a → a
gmapT f = gfoldl k id
          where
              k c x = c (f x)
```

Assim como `gmapT`, o combinador `gmapQ`, definido na Figura 4.5, aplica uma função polimórfica a todos os descendentes imediatos de um valor. Porém, ao contrário de `gmapT`, este novo combinador tem por objetivo realizar *pesquisas genéricas*. Observe que `gmapQ` recebe como parâmetro uma função de tipo

```
(forall b .  Data b ⇒ b → r)
```

que extrai algum valor de tipo `r` do tipo b, e aplica esta função ao valor de tipo `a`, retornando uma lista de todos os valores de tipo `r` que puderam ser obtidos de seus descendentes imediatos.

```
gmapQ :: (∀ b . Data b ⇒ b → r) → a → [r]
gmapQ f = gfoldl k (const id) []
          where
              k c x rs = c (f x : rs)
```

Figura 4.5: Código para o combinador *gmapQ*.

O combinador `gmapM`, mostrado na Figura 4.6, é uma generalização de `gmapT` para tipos monádicos. De fato, `gmapT` pode ser visto como `gmapM` sobre a mônada identidade[LJ03].

```
gmapM :: Monad m ⇒ (∀ b . Data b ⇒ b → m b) → a → m a
gmapM f = gfoldl k return
          where
              k c x = do c' ← c
                         x' ← f x
                         return (c' x')
```

Figura 4.6: Código para o combinador *gmapM*.

Os combinadores `gmapM, gmapT` e `gmapQ` podem ser utilizados para definir uma série de estratégias de alto nível para caminhamento. No trecho de código mostrado a seguir, são definidas duas destas estratégias - `everywhere` (ou *bottom-up*) e `everything`. A primeira aplica uma função de transformação em toda parte da estrutura de maneira *bottom-up*; a segunda é utilizada para descrever consultas. Como pode ser observado em sua definição, a função `everything` processa todos os descendentes de um valor `x`, fornecendo uma lista de resultados que são combinados utilizando a função `foldl`[Jon03].

```
type GenericT = ∀ a. Data a ⇒ a → a
everywhere :: GenericT → GenericT
everywhere f = f . gmapT (everywhere f)

type GenericQ = ∀ a. Data a ⇒ a → r
everything :: (r → r → r) → GenericQ r → GenericQ r
everything k f x = foldl k (f x) (gmapQ (everything k f) x)
```

Figura 4.7: Código dos combinadores everywhere e everything.

Apesar dos combinadores apresentados em [LJ03] fornecerem uma solução prática para expressar algoritmos de maneira genérica, algumas funções genéricas não podem ser expressas simplesmente em termos de caminhamentos. Exemplos clássicos são funções como *igualdade genérica* ou *pretty-printers*, que requerem que duas estruturas sejam percorridas simultaneamente, ou que a estrutura interna do tipo seja exposta. Para isto, Lammel e Jones[LJ04] propuseram uma extensão ao padrão *SYB* que provê suporte a reflexão, acrescentando novas operações à classe `Data`. As operações destinadas a representação interna de tipos de dados e sua manipulação não serão apresentadas aqui, por serem muito próximas de uma representação da sintaxe abstrata de Haskell. Além destas operações para reflexão, neste mesmo trabalho são apresentados novos combinadores, como, por exemplo, uma versão genérica da função *zipWith*[Jon03], que permite a travessia de duas estruturas simultaneamente. Assim

como as versões genéricas da função *map* (`gmapT, gmapQ` e `gmapM`), existem também três versões genéricas da função *zipWith*. Para exemplificar a definição destas funções genéricas equivalentes a *zipWith*, será apresentada e explicada a implementação de `gzipWithQ`, que aplica uma operação de consulta em dois tipos de dados arbitrários. As definições de `gzipWithT` e `gzipWithM` são análogas à de `gzipWithQ` e, portanto, serão omitidas. Antes de apresentar e explicar a implementação de `gzipWithQ`, vamos considerar seu tipo. Para isso, considere os tipos das funções *map* e *zipWith* que estão definidas na biblioteca padrão de Haskell:

```
map     ::      (b→c)   →        [b]   →  [c]
zipWith ::  (a→b→c)   →  [a] → [b]   →  [c]
```

Avaliando o tipo destas duas funções, podemos pensar que *zipWith* é uma versão de *map* para funções que possuem dois argumentos (i.e. funções que possuem um tipo que pode ser unificado com a→b→c). Por analogia, podemos tentar definir `gzipWithQ` como uma versão de `gmapQ` para consultas de dois argumentos. Seguindo esta analogia podemos definir o tipo de `gzipWithQ` a partir de `gmapQ`, conforme mostrado a seguir:

```
gmapQ  ::  Data a
       ⇒   (∀ b .  Data b ⇒ b → r)
       →   a → [r]

gzipWithQ  ::  (Data a1, Data a2)
           ⇒   (∀ b1 b2.  (Data b1, Data b2)) ⇒ b1 → b2 → r)
           →    a1 → a2 → [r]
```

Figura 4.8: Tipos de gmapQ e gzipWithQ.

A função original, `gmapQ f t`, recebe como argumento uma função polimórfica `f` e a aplica a cada descendente imediato do valor `t`, e retornando a lista de todos os elementos de tipo `r` resultantes. De maneira similar, `(gzipWithQ f t1 t2)` aplica a função polimórfica `f` aos correspondentes pares de descendentes imediatos de `t1` e `t2`, retornando uma lista de resultados desta aplicação.

Observando o tipo de `gmapQ`, apresentado na Figura 4.8, podemos perceber que a variável de tipo `a` não está presente no conjunto de variáveis de tipo do primeiro parâmetro de `gmapQ`. Portanto, podemos reescrever o tipo desta função de acordo com o seguinte isomorfismo $\forall\alpha \cdot \tau_1 \rightarrow \tau_2 \equiv \tau_1 \rightarrow \forall\alpha \cdot \tau_2$ (desde que $\alpha \notin \tau_1$):

```
gmapQ  ::   (∀ b.  Data b ⇒ b → r)
       →    (∀ a.  Data a ⇒ a → [r])
```

Usando o sinônimo de tipos `GenericQ`, apresentado na Figura 4.7, o tipo de `gmapQ` poderia ser escrito como `GenericQ r → GenericQ [r]`. Aplicando o mesmo isomorfismo ao tipo de `gzipWithQ`, apresentado na Figura 4.8, e uando o sinônimo `GenericQ`, o tipo de `gzipWithQ` é expresso como `GenericQ (GenericQ r) → GenericQ (GenericQ [r])`. O primeiro argumento de `gzipWithQ` é uma consulta genérica (tipo `GenericQ (GenericQ r)`), que retorna, como resultado, outra consulta genérica (de tipo `GenericQ (GenericQ [r])`). Tal recurso, presente em linguagens funcionais, é chamado de *currying*[2].

A implementação de `gzipWithQ` é aparentemente complexa, uma vez que este combinador deve percorrer duas estruturas diferentes. Porém, o tipo desta função mostra que esta é uma consulta genérica que retorna outra consulta genérica. Portanto, pode-se avaliar `gzipWithQ f t1 t2` em dois passos:

1. Aplica-se `f` a todos os nodos que são descendentes imediatos de `t1` usando `gmapQ` obtendo, assim uma lista de consultas genéricas como resultado.

2. Utiliza-se uma função que aplica todas consultas produzidas no item 1 aos correspodentes descendentes imediatos de `t2`.

Portanto, uma primeira tentativa de definir `gzipWithQ` é:

```
gzipWithQ f t1 t2 = gApplyQ (gmapQ f t1) t2
```

Porém, a expressão `gmapQ f t1` retorna como resultado uma lista de consultas genéricas (onde cada uma possui o tipo $\forall$`a`$\cdot$`a`$\rightarrow$`r`), o que não é permitido em sistemas de tipos predicativos[3]. Para que `gzipWithQ` possua um tipo correto, basta utilizar um tipo intermediário para empacotar (*wrap*) as consultas genéricas. O tipo `GQ`, apresentado a seguir, é responsável por *empacotar* consultas:

```
newtype GQ r = GQ (∀ a . a → r)
```

utilizando este tipo, a definição de `gzipWithQ` fica correta, de acordo com as regras do sistema de tipos de Haskell. Esta definição é mostrada a seguir:

```
gzipWithQ f t1 t2 = gApplyQ (gmapQ (λ x → GQ (f x)) t1) t2
```

A função `gApplyQ`, usada na definição de `gzipWithQ`, utiliza um acumulador de tipo `([GQ r], [r])`, enquanto realiza um *fold* sobre cada sub-termo imediato de um valor, da seguinte maneira: para cada sub-termo imediato é consumido um elemento da lista de consultas (componente `([GQ r]`), enquanto é produzido um elemento da

[2]Currying é uma técnica que permite converter uma função de n argumentos em uma função que recebe um argumento por vez, retornando como resultado uma função de $n-1$ argumentos.

[3]Um sistema de tipo é dito ser predicativo se este não permite que variáveis de tipo sejam instanciadas para tipos polimórficos.

lista de resultados (componente `([r])`. Para obter este comportamento, será utilizado o seguinte combinador `k`:

```
k ::  Data c ⇒ ([GQ r], [r]) → c → ([GQ r], [r])
k ((GQ q):  qs, rs) child = (qs, q child :  rs)
```

No trecho de código anterior, o argumento `c` é o tipo do sub-termo imediato atualmente processado. A função `k` recebe como argumento um acumulador e um sub-termo e retorna outro acumulador como resultado. A operação de *fold* é realizada por uma variação da função `gfoldl`, chamada `gfoldlQ`, apresentada a seguir:

```
newtype R r x = R {unR :: r}
gfoldlQ k z t = unR (gfoldl k' z' t)
              where z' _ = R z
                    k' (R r) c = R (k r c)
```

A definição de `gfoldlQ` é análoga à de `gfoldl`, exceto que a primeira utiliza o tipo intermediário `R`, o tipo específico necessário na definição de `gApplyQ` e o tipo parametrizado de `gfoldl`.

A proposta do padrão *SYB*, descrita em [LJ04, LJ03], permite ao programador escrever funções genéricas que podem ter um comportamento particular para tipos de dados específicos. Porém esta abordagem possui a limitação de que todos os casos específicos têm que ser previstos pelo programador *a priori*, uma vez que estes devem estar presentes na definição da função genérica. Devido a este inconveniente, esta abordagem permite apenas a definição de funções genéricas *fechadas* (i.e. não extensíveis). Em [LP05] os autores utilizaram o mecanismo de classes de tipo para permitir a definição de funções genéricas que possam ser posteriormente estendidas. Para isto, o programador deve definir uma nova classe de tipos para o símbolo que será sobrecarregado e, para cada nova definição, deve ser adicionada uma nova instância desta classe. O comportamento genérico deste símbolo pode ser obtido por uma definição de instância que funcionaria como uma definição *default*. A principal inovação desta proposta é utilizar varíaveis que representam classes. Em Haskell, variáveis em uma restrição ou definição de tipos, não podem representar uma classe. Existe porém uma solução que pode ser escrita no padrão Haskell 98[Hug99], que pode ser utilizada para obtenção do comportamento desejado. A utilização desta técnica possui a vantagem de ser modular, uma vez que instâncias não necessitam estar no mesmo módulo onde sua classe foi definida, e podem ser adicionadas sem a necessidade de alterar o código existente.

A técnica de construção de funções genéricas usando combinadores para a travessia de tipos de dados arbitrários torna difícil a comparação desta abordagem com técnicas baseadas na estrutura algébrica de tipos de dados. Os trabalhos [HLB06,

HL06] mostram que a função `gfoldl`, que é a base da implementação de outras funções do *SYB*, é, na verdade, um *catamorfismo* sobre o tipo de dados `Spine`, que representa a estrutura de construção de valores de um tipo qualquer. Essencialmente, estes trabalhos mostram uma abordagem alternativa para a implementação dos combinadores do *SYB*, que utilizam este tipo de dados. A apresentação das implementações utilizando esta abordagem é similar às definições apresentadas nesta seção. O leitor interessado em mais detalhes desta abordagem para implementação do *SYB* pode consultar os trabalhos [HLB06, HL06].

O padrão *SYB* tem, como principais vantagens, o fato de não requerer que valores sejam convertidos para um tipo de representação estrutural e não requerer especialização, em tempo de compilação, de funções genéricas. Porém, este padrão tem a desvantagem de não possibilitar definir funções genéricas sobre tipos que possuem variáveis de *kind* superior (i.e. *kinds* diferentes de $\star$), uma vez que, para estes tipos, não é possível definir instâncias da classe *Typeable*[HJL06].

Para explicar esta limitação do *SYB*, considere o seguinte tipo de representação, utilizado pela classe `Typeable` e outros dois tipos algébricos:

```
data TypeRep = TypeRep String [TypeRep]
data [a] = [] | a : [a]
data GRose f a = GNode a (f (GRose f a))
```

As instâncias das classes `Data` e `Typeable` para o tipo `[a]` são mostradas no trecho de código seguinte:

```
instance Typeable a ⇒ Typable [a] where
    typeOf x = TypeRep "[a]" [typeOf (get x)]
               where get:: [a] → a
                     get x = undefined


instance Data a ⇒ Data [a] where
    toConstr [] = nilConstr
    toConstr (_:_) = consConstr
    . . . -- outras definições da classe Data
nilConstr = mkConstr listDataType "[]" [] Prefix
consConstr = mkConstr listDataType "(:)" [] Infix
```

Como pode ser visto, a definição é direta e já está presente na implementação do SYB. Os valores `nilConstr` e `consConstr` são utilizados para representar os dois construtores de dados do tipo `[a]`. As instâncias das classes `Data` e `Typeable`, para o tipo `[a]`, necessitam que o tipo, representado pela variável `a`, também pertença a

estas classes (isto é especificado pelas restrições `Typeable a` e `Data a`). O compilador GHC não consegue derivar automaticamente instâncias para o tipo `GRose`, uma vez que, não é possível definir instâncias da classe `Data` para tipos que possuam variáveis de *kind superior*. Considere, a seguinte tentativa de definir instâncias das classes `Data` e `Typeable` para o tipo `GRose`:

```
instance (Typeable1 f) ⇒ Typeable (GRose f a) where
   typeOf x = mkTyConApp (mkTyCon "GRose") [typeOf1 (get x)]
              where
                get :: (GRose f a) -> f a
                get x = undefined

instance (Typeable1 f, Data a, Data (f (GRose f a)),
          Typeable (GRose f a))⇒Data (GRose f a)where
   toConstr x = con1
   dataTypeOf _ = ty
   gunfold k z c = case constrIndex c of
                     1 → k (k (z GNode))
   . . . -- outras definições da classe Data
```

Na definição anterior da classe data, temos a seguinte restrição: `Data (f (GRose f a))`, que informa que deve existir uma instância da classe `Data` para o tipo `f (GRose f a)`. Porém, ao tentarmos definir uma instância que satisfaça esta restrição vemos que não é possível definir algumas operações da classe `Data` para este tipo. Como exemplo desta impossibilidade, considere a tarefa de implementar a função `toConstr`. Esta função retorna um valor que representa o construtor de dados utilizado em um determinado valor. A implementação de `toConstr` realiza um casamento de padrão sobre o valor para determinar qual foi o construtor utilizado. Porém, em valores de tipo `f (GRose f a)`, não é possível determinar qual o construtor de dados, do construtor de tipos representado pela variável `f`, foi utilizado para formar um valor deste tipo. Por isto, SYB não permite definir funções genéricas sobre tipos que são parametrizados sobre um construtor de tipos, como `GRose`.

4.2.1 Exemplos

Para exemplificar o uso desta abordagem para a eliminação de *boilerplate code*, considere o problema de atualizar os salários de todos os funcionários de uma empresa que é representada em um programa por meio de um tipo de dados mutuamente recursivo

(exemplo retirado de [LJ03]). O tipo de dados, apresentado na Figura 4.9 representa a estrutura organizacional desta empresa:

```
data Company = C [Dept]
data Dept = D Name Manager [SubUnit]
data SubUnit = PU Employee | DU Dept
data Employee = E Person Salary
data Person = P Name Address
data Salary = S Float
type Manager = Employee
type Name = String
type Address = String
```

Figura 4.9: Definição de tipos algébricos para representar uma empresa.

A função de atualização dos salários de todos os funcionários possui uma implementação direta em Haskell, que pode ser encontrada em [LJ03], mas tal abordagem possui o inconveniente de que praticamente todo o código escrito é destinado à travessia do tipo de dados. Somente a função `incS` (apresentada a seguir), que altera o valor de salário, em termos de uma dada porcentagem, realiza algum tipo de *ação*.

```
incS :: Float → Salary → Salary
incS k (S s) = S(s * (1 + k))
```

Para aumentar o salário de todos os funcionários desta empresa, sem a utilização de *boilerplate code*, basta usar `incS` em um caminhamento por toda a estrutura, tal como na definição de `increaseSalary`, apresentada a seguir. Na definição de `increaseSalary` é usado o combinador `everywhere`, que aplica uma função em todas as partes de um valor. A função `incS` é combinada com `mkT`, resultando em uma função que modifica apenas valores do tipo `Salary`, comportando-se como a função identidade sobre todos os outros tipos.

```
increaseSalary :: Float → Company → Company
increaseSalary k = everywhere (mkT (incS k))
```

Figura 4.10: Definição de increaseSalary usando SYB.

Ainda considerando o tipo de dados que representa a estrutura organizacional de uma empresa, suponha que se deseja definir uma função para calcular o total gasto com sua folha de pagamento. Tal função deveria percorrer toda a estrutura de dados da empresa, somando o valor do salário de cada funcionário. Para isso, é necessário a

criação de uma consulta para a extração do valor numérico (de tipo `Float`) de `Salary`, tal como a função `billS` a seguir:

```
billS :: Salary → Float
billS (S f) = f
```

Assim como no exemplo anterior, há a necessidade de estender esta função para operar sobre tipos distintos do tipo `Salary`. Como esta função consiste de uma consulta, sua extensão, é feita utilizando a função `mkQ`. A função `salaryBill`, calcula o total gasto com o pagamento de salários, utilizando o combinador `everything`:

```
salaryBill :: Company → Float
salaryBill = everything (+) (mkQ 0 billS)
```

Estes dois exemplos ilustram como *SYB* pode ser utilizado para a construção de funções que percorrem tipos de dados complexos. Mas, a utilização do padrão *SYB* não se resume à eliminação de código de travessia de estruturas de dados, este pode ser utilizado para a definição de funções genéricas clássicas, como igualdade genérica, *pretty-printers* e *parsers*.

A igualdade genérica pode ser facilmente definida em termos do combinador `gzipWithQ`, conforme mostrado no trecho de código a seguir:

```
geq1 :: GenericQ (GenericQ Bool)
geq1 x y = toConstr x == toConstr y && and (gzipWithQ geq1 x y)
geq :: Data a ⇒ a → a → Bool
geq = geq1
```

O funcionamento desta definição é simples. Primeiramente, verifica-se se os valores `x` e `y` utilizam o mesmo construtor de dados; em caso positivo, os descendentes de `x` e `y` são compostos em pares e comparados em relação a igualdade, usando `geq1`. Como resultado, é retornada uma lista de valores booleanos que, em seguida, é submetida à função `and` :: $[Bool] \rightarrow Bool$[4], para produzir o resultado desejado.

A função `gshow` é utilizada para realizar a conversão de um valor qualquer (cujo tipo seja instância da classe `Data`) e convertê-lo em um string. Assim como `geq`, `gshow` possui uma implementação simples:

```
gshow :: Data a ⇒ a → String
gshow t = "(" ++ showConstr (toConstr t)
              ++ concat (intersperse " " (gmapQ gshow t))
              ++ ")"
```

[4] A função `and` retorna o resultado da conjunção da lista de valores do tipo `Bool` fornecida como parâmetro.

A função `toConstr` retorna o construtor de dados utilizado para construir o valor `t` e `showConstr` converte um valor de tipo `Constr` em um string. Cada um dos descendentes imediatos de `t` é também processado usando `gshow`, produzindo uma lista de strings como resultado. Esta lista é então submetida às funções `intersperse` e `concat`[5] para produzir a string que representa o valor `t`.

Aparentemente, a tarefa de construir um valor a partir de um string é apenas o inverso da operação de *pretty-printing*. Observando a definição de `gshow`, pode-se imaginar que uma função que realiza a conversão de string em valores de outro tipo qualquer deveria utilizar uma função inversa de `toConstr`. Considere então a existência, na classe `Data`, de uma função `fromConstr :: Constr → a`, que gera um valor de um tipo de dados a partir de seu construtor. Porém, somente um construtor de dados não fornece informação suficiente para construir um valor: é necessário saber quais são os argumentos aos quais este construtor deve ser aplicado. A solução para este problema é apresentada em [LJ04]. Neste trabalho, os autores propõe a passagem de uma função monádica para permitir que a informação sobre a geração dos argumentos deste construtor esteja disponível em todos os pontos do processo de *parsing*.

O código para uma função de *parsing* genérica, utilizando o *SYB* é:

```
gread :: Data a ⇒ String → Maybe a
gread input = runDec input readM
```

A função `runDec` executa um *parser* sobre uma entrada particular, descartando o estado final e a parte não consumida da entrada, retornando o resultado do *parsing*. Por sua vez, a função `readM`, tem a finalidade de realizar as seguintes ações:

1. Realizar o *parsing* da representação do construtor de dados deste valor.

2. Usar a função `fromConstrM` para chamar sucessivamente `readM`, para realizar o *parsing* de cada um dos argumentos do construtor de dados e, a partir deste resultado, construir o valor utilizando o construtor identificado no Item 1.

Para efetuar o *parsing* de um construtor é necessário saber a qual tipo de dados este construtor pertence. Isto pode ser resolvido utilizando a função `dataTypeOf`, que possui o tipo:

```
dataTypeOf :: a → DataType
```

[5] As funções `intersperse` e `concat` estão definidas na biblioteca padrão de Haskell[Jon03]. `concat::[[a]]→ [a]` concatena os elementos de uma lista de listas e `intersperse :: a → [a] → [a]` insere seu primeiro argumento entre cada par de elementos consectivos de seu segundo argumento.

Esta função retorna, como resultado, uma representação do tipo de dados de um determinado valor. A utilização desta função requer um valor do tipo de dados cuja representação é desejada. Porém, este valor ainda está sendo construído pelo processo de *parsing*. Este problema é facilmente resolvido pela avaliação *lazy* utilizada em Haskell: para isto, define-se uma função `unDec` de tipo:

```
unDec ::  DecM a → a,
```

que jamais é avaliada, sendo sua utilidade mostrar para o compilador qual dicionário da classe `Data` deve ser utilizado para resolver a sobrecarga. O código para a função `readM` é:

```
readM :: Data a ⇒ DecM a
readM = read_help
         where
              read_help = do
                            let
                                ty = dataTypeOf (unDec read_help)
                            constr ← parseConstr ty
                            fromConstrM readM constr
              unDec :: DecM a → a
              unDec = undefined
```

Toda a estratégia para realizar a conversão de um string em um tipo de dados qualquer é encapsulada pelo tipo `DecM`[6], que utiliza uma abordagem padrão para a definição de mônadas de *parsing*[HM98]:

```
newtype DecM a = D (String → Maybe (String, a))
```

A última definição necessária para completar a implementação de `gread` é a da função `parseConstr`, apresentada a seguir:

```
parseConstr :: DataType → DecM Constr
parseConstr ty = D (λ s → match s (dataTypeConstrs ty))
                  where
                      match :: String → [Contr] → Maybe (String, Contr)
                      match _ [] = Nothing
                      match input (con:cons)
                                          | take (length s) input == s
                                            = Just (drop (length s) input, con)
```

[6]DecM - **Dec**oder **M**onad

```
                                    | otherwise = match input cons
                                   where s = showConstr con
```

No trecho de código anterior, a função `parseConstr` utiliza `dataTypeConstrs`, a qual retorna uma lista de todos os construtores presentes em um tipo de dados. Primeiramente, `parseConstr` tenta realizar o casamento de cada construtor com o início da entrada, retornando o construtor encontrado e o restante da string, ou retornando `Nothing` caso não seja possível realizar este casamento.

A função `fromConstrM` está presente na classe `Data`; o seu tipo é apresentado no trecho de código seguinte.

```
class Typeable a ⇒ Data a where
    ...
    fromConstrM :: (Monad m, Data a) ⇒ (∀ b. Data b ⇒m b) → Constr
     → m a
    ...
```

As definições desta função são extremamente simples, como pode ser observado no trecho de código a seguir, que implementa `fromConstrM` para listas:

```
instance Data a ⇒ Data [a] where
       fromConstrM f con = case constrIndex con of
                                  1  → return []
                                  2  → do a  ← f
                                          as ← f
                                          return (a:as)
```

Nesta definição, o parâmetro `f` é responsável por realizar o *parsing* da entrada que está sendo processada, obtendo os argumentos do construtor passado como parâmetro (`con`) e retornando o valor construído a partir do construtor e de seus argumentos. A identificação do construtor utilizado para formar o valor retornado é feita pela função `constrIndex :: Constr → ConIndex`, que, a partir de um valor do tipo `Constr`, retorna um inteiro que identifica unicamente um construtor dentro do conjunto de construtores de dados presentes em um tipo algébrico.

Assim como `gmapQ` e `gmapT` são definidas em termos de uma função mais geral − `gfoldl` − `fromConstrM` é definida em termos de uma outra função mais geral chamada `gunfold`, que é *dual* de `gfoldl`. O tipo de `gunfold` e um exemplo de sua definição para listas são apresentados a seguir, na Figura 4.11.

Se o argumento `d :: Constr` é o valor que representa todas as informações disponíveis sobre um determinado construtor de dados C, que recebe n argumentos, então a

```
class Typeable a ⇒ Data a where
    gunfold :: (∀ b r. Data b⇒c (b→ r)→c r) →
               (∀r.r → c r) → Constr →  c a
    ...
instance Data a Data [a] where
    gunfold k z c = case constrIndex c of
                          1 →z []
                          2 →k (k (z (:)))
                          _ →error "gunfold"
```

Figura 4.11: Tipo e uma instância para a função *gunfold*.

expressão `gunfold app c d` é avaliada como: `app(...(app(c C)) ...)`. Assim, o parâmetro `app` é aplicado n vezes a c C. Observe que, na definição de `gunfold` para listas, presente na Figura 4.11, temos que o parâmetro `k` é aplicado a cada construtor de dados do tipo lista um número de vezes idêntico ao número de argumentos que este construtor de dados possui (nenhuma vez para o construtor `[]` e duas vezes para o construtor `(:)`).

Utilizando `gunfold`, a definição de `fromConstrM` seria:

```
fromConstrM f = gunfold k z
                where
                     k c = do
                             c' ← c
                             b  ← f
                             return (c' b)
                     z = return
```

No trecho de código anterior o argumento `z`, presente na expressão `(gunfold k z)`, é utilizado para transformar um construtor de dados em uma função monádica. A função `k` é utilizada para realizar o processamento dos argumentos que serão aplicados ao construtor de dados utilizado, para construir o valor em questão.

4.3 Generic Haskell

Generic Haskell[dW02, Löh04, HJ02b, HJ02a] é uma extensão da linguagem *Haskell*, projetada especificamente para o desenvolvimento de programas genéricos. Para isto, adiciona à linguagem novas construções, que permitem a definição de valores indexados por tipos, tipos indexados por *kinds* e tipos indexados por tipos. Esta seção visa apresentar a linguagem *Generic Haskell* e como esta pode ser utilizada para definir

funções genéricas. Para aspectos relativos ao processo de compilação desta linguagem, o leitor interessado pode consultar [dW02, Löh04].

4.3.1 Parênteses especiais

Definições indexadas por tipo possuem um argumento de tipo que é definido entre "parênteses"especiais. Os parênteses {| |} são utilizados para incluir um argumento de tipo em um tipo indexado por tipos. Apesar de não serem mais necessários, a linguagem provê suporte, para fins de compatibilidade, aos parênteses {[ ]}, que permitem incluir um argumento que é um *kind*, que pode ser distinto de $\star$, para tipos indexados por *kinds*. Neste texto, será utilizada a notação presente em diversos trabalhos sobre *Generic Haskell*[Löh04, HJ02b, HJ02a], onde parênteses especiais são representados por $\langle\ \rangle$.

4.3.2 Funções indexadas por tipos e parâmetros de tipos

Funções indexadas por tipos são definidas como funções sobre tipos ou sobre a estrutura algébrica de construtores de tipos. Uma função indexada por tipo consiste em um ou mais casos, que definem esta função por casamento de padrão sobre o argumento de tipo. Como um primeiro exemplo, considere a tarefa de definir uma função `add` sobre os tipos `Bool`, `Int` e `Char`, onde o comportamento para o tipo `Bool` é idêntico ao da função ($\vee$), para `Int` esta função se comporta como a adição sobre inteiros e para o tipo `Char` ela gera um novo caractere somando os valores do código ASCII dos parâmetros. A definição desta função, em Generic Haskell, pode ser feita como a seguir:

```
add ⟨Bool⟩ = (||)
add ⟨Int⟩ = (+)
add ⟨Char⟩=(λx y→chr(add ⟨Int⟩(ord x) (ord y))
```

Aparentemente, esta definição se parece com a definição de três funções distintas, uma para cada um dos tipos `Bool, Int` e `Char`. A principal diferença desta definição em relação à definição de três funções distintas é o seu tipo. O tipo de `add` é:

$$\texttt{add}\ \langle a :: \star \rangle :: a \to a \to a\,.$$

A notação `add` $\langle a :: \star \rangle$ significa que a função `add` possui um argumento de tipo, que é representado pela variável `a`, que deve ser de *kind* $\star$. Cada uma das equações desta definição utiliza um padrão de tipo que deve ser correspondente ao tipo de `add` (observe que os tipos `Bool, Int` e `Char` possuem *kind* $\star$). De acordo com esta definição, as seguintes expressões são válidas:

O resultado da avaliação de cada uma dessas expressões seria `True`, `18` e `'a'`, respectivamente.

```
add ⟨Bool⟩ True True
add ⟨Int⟩ 15 3
add ⟨Char⟩ 'A' ' '
```

Figura 4.12: Definição da função `add`.

Generic Haskell permite que padrões de equações sejam definidos utilizando construtores de tipos de *kind* arbitrário. Como exemplo, considere o problema de definir a função `add` sobre o tipo `Maybe a`. Evidentemente, para realizar a adição de valores do tipo `Maybe`, deve existir uma função que realiza a soma de valores do tipo `a`, tratando o valor `Nothing` como um elemento nulo da operação de adição para `Maybe`. As equações que definem `add` para o tipo `Maybe a` são:

```
add ⟨Maybe a⟩ Nothing _ = Nothing
add ⟨Maybe a⟩ _ Nothing = Nothing
add ⟨Maybe a⟩ (Just x) (Just y) = Just (add ⟨a⟩ x y)
```

Evidentemente, que as possibilidades de utilização de parâmetros de tipos não se restringem a construtores de *kind* $\star \rightarrow \star$ (como `Maybe`). Podem-se utilizar construtores de tipos de *kind* arbitrário. Como exemplo de um construtor de tipos de *kind* superior considere a definição de `add` para pares (o contrutor de tipos (,) possui *kind* $\star \rightarrow \star \rightarrow \star$):

```
add⟨(a,b)⟩(r,s)(x,y) = (add⟨a⟩ r x, add⟨b⟩ s y)
```

Considere, novamente, as equações que definem `add` para `Maybe`. Observe que a terceira equação possui uma *restrição de dependência* que é representada pela ocorrência da chamada de `add` sobre a variável de tipo `a`. Dependências surgem na definição de uma função indexada por tipo se se sua definição usa uma função indexada por tipo que possui uma variável como argumento. A existência de uma dependência na definição de uma função indexada por tipo reflete-se no tipo desta função. As equações para a definição de `add` para o tipo `Maybe` possuem o tipo:

$$add\langle \text{Maybe a}\rangle :: \forall a :: \star.(add\langle a\rangle :: a \rightarrow a \rightarrow a) \Rightarrow \text{Maybe a} \rightarrow \text{Maybe a} \rightarrow \text{Maybe a}\ .$$

O tipo $add\langle a\rangle :: a \rightarrow a \rightarrow a$, especificado entre parênteses, representa a restrição de dependência das equações que definem `add` para `Maybe`. Isto significa que, para utilizar o símbolo `add` para valores do tipo `Maybe` deve haver uma definição para `add` de tipo $a \rightarrow a \rightarrow a$, no mesmo contexto onde `add` será usado.

Restrições de dependências são equivalentes a restrições de classe[WB89] em Haskell e a restrições presentes em tipos polimórficos do sistema CT[Vas04, CF99a, CaLFN07].

Apesar dos parâmetros de tipos utilizados por *Generic Haskell* na definição de equações fornecerem um meio prático para definição de funções genéricas, este possuem algumas restrições de utilização. A restrição para a definição de padrões de tipo é que todas as variáveis de tipo utilizadas devem ser distintas, ou seja, o padrão ⟨(a,a)⟩ é inválido. Uma outra restrição existente para a definição de padrões de tipos é que estes não podem ser **aninhados**, i.e., construtores de tipos devem ser aplicados apenas a variáveis de tipos, não sendo permitido que outros construtores sejam utilizados em lugar de variáveis. Exemplos de parâmetros inválidos segundo esta restrição são: ⟨[Maybe a]⟩ e ⟨(Either Int Bool,a)⟩.

4.3.3 Definições de Funções por Indução Estrutural

Nos exemplos anteriores, foram apresentadas algumas características de *Generic Haskell*, como funções indexadas por tipos, padrões de tipos e dependências. Porém, todas as definições apresentadas podem ser definidas utilizando *polimorfismo de sobrecarga*. Nesta seção mostramos como se pode definir funções baseadas na estrutura algébrica de tipos de dados Haskell, i.e., funções que utilizam *polimorfismo estrutural*.

Conforme apresentado na Seção 4.1.1, todo tipo de dados Haskell pode ser representado como uma soma de produtos. Sendo assim, é possível definir um tipo de dados cujos valores capturam a representação estrutural de todo tipo Haskell. O tipo de dados apresentado na Seção 4.1.1 é utilizado pelo compilador de *Generic Haskell* para a representação interna de tipos Haskell.

Considere, novamente, a definição da função indexada por tipo `add`. Nas definições anteriores, foram apresentadas equações para tipos `Int, Bool, Char, Maybe` e pares. Suponha que agora se deseja definir esta mesma função sobre listas. A equação que definiria add para listas seria:

```
add ⟨[a]⟩x y
            | length x == length y = map (uncurry(add⟨a⟩)) (zip x y)
            | otherwise = error "As listas devem ser do mesmo tamanho!"
```

Como pode ser visto, esta definição é simples e direta. O único inconveniente desta abordagem é que, para cada novo tipo de dados para o qual se deseje implementar a função `add`, um novo conjunto de equações deve ser adicionado.

Para contornar esta situação, pode-se definir a função `add` por indução sobre a estrutura algébrica de tipos de dados. A definição de `add` utilizando este recurso é:

Considerando os tipos de dados de representação estrutural apresentados na Seção 4.1.1 e que os tipos `Maybe a` e `[a]` são definidos como:

```
data Maybe a = Nothing | Just a
```

```
add ⟨Bool⟩ = (||)
add ⟨Int⟩ = (+)
add ⟨Char⟩ = (λx y → chr(add ⟨Int⟩ (ord x)(ord y)))
add ⟨Unit⟩ Unit Unit = Unit
add ⟨Prod a b⟩ (a1 × b1) (a2 × b2) = (add⟨a⟩ a1 a2) × (add⟨b⟩ b1 b2)
add ⟨Sum a b⟩ (Inl a1) (Inl a2) = add⟨a⟩ a1 a2
add ⟨Sum a b⟩ (Inr b1) (Inr b2) = add⟨b⟩ b1 b2
add ⟨Sum a b⟩ _ _ = error "Os parâmetros devem ser do mesmo tipo!"
```

Figura 4.13: Definição da função `add` usando indução estrutural.

```
data [a] = [ ] | a :  [a]
```

Os seguintes tipos são isomórficos a `Maybe a` e `[a]`, respectivamente:

```
type MaybeRep a = Sum Unit a
type ListRep a = Sum Unit (Prod a [a])
```

Observe que a definição de `add` é feita sobre os tipos estruturais `Sum, Prod` e `Unit`. Para esta definição genérica funcionar efetivamente sobre tipos `Maybe a` e `[a]`, valores destes tipos devem ser convertidos para valores dos tipos de representação estrutural. Funções que realizam esta conversão são automaticamente geradas pelo compilador de *Generic Haskell*, devido a sua simplicidade de implementação.

4.3.4 Tradução por Especialização

Funções indexadas por tipos são compiladas para código Haskell usando um processo chamado de *especialização*. Nesta seção apresentaremos uma breve introdução à tradução de funções escritas em Generic Haskell. Primeiramente, serão dadas algumas definições e, em seguida, mostraremos como traduzir uma função genérica por meio de um exemplo. Encerramos esta seção apresentando algumas críticas ao processo de especialização utilizado por Generic Haskell. Não abordaremos todo o processo de tradução realizado por Generic Haskell por este estar fora do escopo deste trabalho.

4.3.4.1 Definições

Para apresentarmos o processo de especialização de funções indexadas por tipos, utilizaremos as definições apresentadas em [Löh04].

Um primeiro conceito é o de *assinatura*. A *assinatura* de uma função genérica é definida como sendo a lista de tipos que aparecem em padrões de tipos que definem cada uma das equações desta função[Löh04]. Como exemplo, considere a definição de `add` presente na figura 4.12, sua *assinatura* é {`Int`, `Bool`, `Char`}

Uma declaração de função indexada por tipo é traduzida em um grupo de declarações com diferentes nomes, onde cada equação correspondente a um tipo é compilada em uma definição. O processo de tradução de funções genéricas utiliza a função *cp* que recebe como parâmetro uma variável e um nome de tipo e retorna um novo nome diferente de seus parâmetros. Por exemplo, *cp(add, Bool)* poderia fornecer como resultado *add_Bool*.

O processo de tradução utiliza a *assinatura* e o identificador de uma função genérica para a geração do código Haskell correspondente. Dá-se o nome de *componente* ao resultado do processo de tradução de uma única equação de uma função indexada por tipos. Como exemplo, considere novamente a função `add` apresentada na figura 4.12. O *componente* correspondente a equação cujo parâmetro de tipo é `Bool` é:

cp(add, Bool) `= (||)`

Finalmente, chamamos o processo de traduzir uma chamada de uma função indexada por tipos de *especialização*, porquê substitui-se a chamada para a função que depende de um argumento de tipo pela chamada para a versão especializada desta função. Ainda considerando a função `add` (figura 4.12), observe a equação que define esta função para o tipo `Char`:

`add ⟨Char⟩=(λx y→chr(add ⟨Int⟩(ord x) (ord y))`

Nesta equação é feita uma chamada à definição de `add` para valores do tipo `Int`. Neste caso, esta chamada deve ser convertida para o identificador correspondente ao *componente* gerado para o tipo `Int`. O resultado da tradução seria:

cp(add, Char) `=(λx y→chr(` *cp(add, Int)* `(ord x) (ord y))`

O resultado da tradução substitui a chamada `add ⟨Int⟩` por *cp(add, Int)*, realizando a especialização desta chamada da função `add`.

O código completo gerado pelo compilador de Generic Haskell, para a função `add` é apresentado na figura 4.14.

4.3.4.2 Conclusão

A tradução por especialização possui como principal vantagem que argumentos de tipos são eliminados do código, e instâncias especializadas são usadas sobre cada um dos diferentes tipos que compõe esta função. Para funções indexadas por tipos não genéricas (i.e. que não usam de indução estrutural), não possuem o problema de performance causado por conversões entre tipos de dados e sua representação estrutural. Chamadas para este tipo de funções indexadas por tipos são substituídas diretamente por chamadas às instâncias especializadas geradas em tempo de compilação.

```
cp(add, Int) :: Int → Int → Int
cp(add, Int) = (+)

cp(add, Bool) :: Bool → Bool → Bool
cp(add, Bool) = (||)

cp(add, Char) :: Char → Char → Char
cp(add, Char) x y = cp(add, Int) (ord x) (ord y)
```

Figura 4.14: Resultado da tradução da função `add`.

Uma desvantagem da especialização é o fato que funções indexadas por tipos não são *cidadãos de primeira classe*. Em Generic Haskell, o nome de uma função indexada por tipo não é por si só uma expressão válida [Löh04]. Tal restrição é fundamental para a permitir a especialização de maneira simples, pois com funções genéricas de primeira classe é difícil determinar quais componentes deverão ser especializados.

Outra desvantagem do processo de especialização é que o código gerado para componentes de funções indexadas por tipos que usam representação estrutural não é eficiente: valores são convertidos de seus tipos originais para a sua respectiva representação estrutural e vice-versa, durante todo o tempo de execução. Sempre que uma função que utiliza indução estrutural é chamada faz-se a conversão de / para representação estrutural, o número destas conversão é consideravelmente alto se a definição da função genérica é recursiva. Em [Löh04] são citadas técnicas experimentais para a otimização de funções genéricas que utilizam representação estrutural. Tais otimizações são baseadas em avaliação parcial e apresentaram resultados promissores descritos em [Löh04].

4.3.5 Tipos Algébricos Indexados por Tipos

Funções indexadas por tipos são funções que possuem um tipo como parâmetro e podem acessar a estrutura de seu argumento de tipo em sua definição, podendo, assim, assumir comportamentos diferenciados para diferentes tipos[Löh04].

Este mecanismo pode ser utilizado não apenas em definições de funções, mas também em definições de tipos algébricos. Um *Tipo Algébrico Indexado por Tipos* (*Type-indexed data type*) é um tipo algébrico com um argumento de tipo, que é utilizado para definir a estrutura deste novo tipo, ou seja, tipos algébricos indexados por tipos podem ter diferentes implementações para diferentes argumentos.

Um exemplo de tipo algébrico indexado por tipos são as chamadas *Tries*, que são freqüentemente utilizadas como índices para textos. Uma árvore de busca digital, ou *trie*, é uma árvore de busca que utiliza a estrutura das chaves para organizar a

informação eficientemente[RH02]. Isto quer dizer que é necessário implementar uma nova *trie* para cada tipo de chave utilizada. Por exemplo, considere a seguinte implementação de uma *trie* que utiliza chaves do tipo `String`:

```
type FMapChar v = [(Char, v)]
data FMapString v = Trie_String (Maybe v)
                                (FMapChar (FMapString v))
```

Neste exemplo o prefixo `FMap`, presente nos contrutores, é uma abreviação para mapeamento finito. O primeiro componente do construtor `Trie_String` contém o valor associado a uma `String` vazia e seu tipo é `Maybe v`, ao invés de `v`, devido ao fato de que há a possibilidade de uma `String` não fazer parte do domínio das chaves desta *trie*. Neste caso, o valor deste componente é igual a `Nothing`. O segundo componente do construtor `Trie_String` realiza o mapeamento de uma *trie* de chaves do tipo String utilizando uma *trie* cujas chaves são caracteres. A partir destas definições, é possível definir funções para a busca de strings:

```
lookupChar :: ∀v.Char→FMapChar v→Maybe v
lookupChar c [ ] = Nothing
lookupChar c ((c', v):xs)
                      | c < c' = Nothing
                      | c == c' = Just v
                      | c > c' = lookupChar c xs

lookupString :: ∀v.String→FMapString v→Maybe v
lookupString [ ] (Trie_String tn tc) = tn
lookupString (x:xs)(Trie_String tn tc) =
                                (lookupChar x ◇ lookupString xs) tc
```

A definição de `lookupChar` é direta. Esta função percorre seqüencialmente a lista de associações `FMapChar` e retorna o valor `Nothing`, caso a chave fornecida não esteja presente neste mapeamento. A função `lookupString` realiza a busca de uma determinada string em uma estrutura `FMapString`. A primeira equação da definição de `lookupString` trata a possibilidade de buscar uma string vazia em uma *trie*. A última equação desta definição, realiza a busca de uma string não vazia `(x:xs)` em dois passos:

1. Primeiramente, busca-se o caractere `x` na estrutura `FMapChar`, presente no segundo componente do construtor `Trie_String`, obtendo como resultado um valor de tipo `Maybe Trie_String`.

2. Caso o resultado do primeiro passo seja diferente de `Nothing`, é feita a chamada recursiva a `lookupString`, passando como argumentos o restante da chave (`xs`) e a sub-árvore `tc`.

Para obter este seqüenciamento, e prover um tratamento adequado para o valor `Nothing`, é utilizada a composição monádica para o tipo `Maybe`, representada por $\diamond$:

```
(◇) ::  ∀ a b c.  (a → Maybe b) → (b → Maybe c) → a → Maybe c
(f ◇ g) = case f a of
                  Nothing → Nothing
                  Just b → g b
```

Antes de apresentar a definição de uma *trie* utilizando a sintaxe de Generic Haskell para tipos algébricos indexados por tipos, considere o seguinte exemplo, de uma *trie* cujas chaves são árvores binárias em que caracteres são armazenados nas folhas. Esta árvore é representada pelo seguinte tipo algébrico:

```
data Bush = Leaf Char | Fork Bush Bush
```

E a *trie* que utiliza como chaves valores do tipo `Bush` e sua correspondente função de busca é mostrada a seguir:

```
data FMapBush v = Trie_Bush (FMapChar v) (FMapBush (FMapBush v))


lookupBush :: ∀ v .Bush → FMapBush v → Maybe v
lookupBush (Leaf c) (Trie_Bush tl tf) = lookupChar c tl
lookupBush (Fork bl br) (Trie_Bush tl tf) =
                          (lookupBush bl ◇ lookupBush br) tf
```

Novamente, temos dois componentes, um representando valores construídos por `Leaf`, e um componente para valores construídos utilizando `Fork`. É fácil perceber que não apenas as funções de busca, mas também os tipos algébricos para *tries* são instâncias de um mesmo padrão genérico[RH02].

Para perceber os padrões genéricos envolvidos nas definições dos tipos algébricos e de funções de busca, considere as seguintes representações estruturais dos tipos `String` e `Bush` e de suas respectivas tries:

```
type StringRep = Sum (Con Unit) (Con (Prod Char [Char]))
type FMapStringRep v = Prod (Con Maybe) (Prod FMapChar FMapString)


type BushRep v = Sum (Con Char) (Con (Prod Bush Bush))
type FMapBushRep v = Prod (Con FMapChar) (Prod FMapBush FMapBush)
```

É evidente que existem isomorfismos entre a representação estrutural de um tipo e a representação estrutural de sua respectiva trie. Estes isomorfismos são conhecidos como as leis exponenciais (*laws of the exponentials*)[RH02]:

$$\begin{aligned} 1 \rightarrow_{fin} v &\cong v \\ (t_1 + t_2) \rightarrow_{fin} v &\cong (t_1 \rightarrow_{fin} v) \times (t_2 \rightarrow_{fin} v) \\ (t_1 \times t_2) \rightarrow_{fin} v &\cong t_1 \rightarrow_{fin} (t_2 \rightarrow_{fin} v) \end{aligned}$$

Nestas definições, $t \rightarrow_{fin} v$ representa o tipo de mapeamentos finitos de t para v. Por meio destes isomorfismos, é possível definir um tipo algébrico indexado por tipos `FMap⟨t⟩ v`, onde cada um dos isomorfismos é utilizado em uma equação que define a estrutura do tipo algébrico, indexado por tipos, em termos do seu parâmetro de tipo. A definição de `FMap⟨t⟩ v` é mostrada a seguir:

```
type FMap⟨t::⋆⟩::⋆ → ⋆
type FMap⟨Unit⟩v = Maybe v
type FMap⟨Char⟩v = FMapChar v
type FMap⟨Sum t1t2⟩v = Prod (FMap⟨t1⟩ v) (FMap⟨t2⟩ v)
type FMap⟨Prod t1t2⟩v = FMap⟨t1⟩ (FMap⟨t2⟩ v)
```

Observe que a definição de um tipo algébrico indexado por tipos é similar à de uma função indexada por tipos. Cada equação que define o tipo algébrico fornece os parâmetros necessários para a especialização, deste tipo para cada um dos casos (tipos primitivos, somas, produtos).

Tipos algébricos indexados por tipos representam uma importante extensão presente em Generic Haskell. Contudo, é possível representar tipos indexados por tipos utilizando classes de tipos e dependências funcionais[RH02, Jon00]. Além disso, em [CKJM05] é apresentada uma extensão ao sistema de classes de tipos, chamada tipos associados (*associated types*), que permite a definição de tipos algébricos indexados por meio de tipos utilizando classes de tipos e instâncias.

4.3.6 Redefinição Local

Na Seção 4.3.2, foram apresentadas as chamadas restrições de dependência. Uma função indexada por tipos pode depender de outras funções indexadas por tipos. Dependências (restrições de dependência) são causadas sempre que uma função indexada por tipo é usada na definição de outra função, utilizando um parâmetro que contêm uma variável de tipo. Quando tal fato ocorre, uma restrição de dependência deve ser adicionada ao tipo da função indexada por tipo que está sendo definida.

Dependências podem ser vistas como dicionários de sobrecarga[WB89], que informam como deve ser realizada uma operação genérica sobre um tipo específico[Löh04].

Redefinições locais são um recurso oferecido por Generic Haskell que permitem ao programador estender ou redefinir as restrições de dependência, sobre uma determinada função indexada por tipo, em um ponto específico do programa.

Como exemplo da utilização deste recurso, considere o seguinte exemplo que codifica uma função identidade genérica:

```
id⟨a::⋆⟩ ::  (id) ⇒ a → a
id⟨Int⟩ x = x
id⟨Char⟩ x = x
id⟨Unit⟩ x = x
id⟨Sum a b⟩ (Inl x) = Inl(id⟨a⟩ x)
id⟨Sum a b⟩ (Inr x) = Inr(id⟨b⟩ x)
id⟨Prod a b⟩ (x × y) = (id⟨a⟩ x)×(id⟨b⟩ y)
```

Figura 4.15: Definição da função identidade genérica.

É evidente que a função identidade é parametricamente polimórfica. Na definição acima, faz-se a aplicação desta função em toda a estrutura de um tipo qualquer. Aparentemente, esta definição genérica não possui utilidade alguma, uma vez que a função identidade tem um comportamento diferenciado de acordo com a estrutura do tipo a que é aplicada. Veja que as equações que definem os casos de `id` para produtos e somas introduzem uma restrição de dependência desta função em relação a ela própria. Tal implementação possui uma vantagem: pode-se utilizar redefinição local para alterar o comportamento desta função. Para isto, considere a tarefa de implementar uma função que dobre todos os valores numéricos em uma lista. Esta tarefa pode ser realizada utilizando uma redefinição local da função `id`. O trecho de código que realiza esta redefinição é mostrado a seguir:

```
double x = let id⟨a⟩ = (* 2) in id⟨[a]⟩ x
```

Nesta definição, a expressão `let id⟨a⟩ = (* 2)` redefine o comportamento de `id` para valores do tipo `a`. Neste caso, a expressão anterior permite que a restrição de dependência de `id` seja resolvida, e esta função assume o comportamento especificado pela expressão `let` anterior.

4.3.7 Definições Padrão

Durante o desenvolvimento de um programa, é comum implementar diversas funções genéricas com pequenas, sejam estas variações acréscimos ou redefinições. Visando solucionar este problema, Generic Haskell oferece um recurso chamado *Definições Padrão* (*default cases*), que permite que uma função genérica seja definida *herdando*

as equações que definem uma outra função, e adicionando ou atualizando equações, conforme necessário.

Como exemplo de utilização deste recurso, considere a tarefa de implementar a igualdade insensitiva de caracteres (i.e. não diferenciando caracteres maiúsculos de minúsculos), que podem estar armazenados em estruturas de dados arbitrárias. Para implementar tal função, basta redefinir a equação que corresponde ao tipo `Char`, conforme mostrado a seguir:

```
equal ⟨a::⋆⟩ ::  (equal⟨a⟩) ⇒ a → a → Bool
equal ⟨Int⟩ = (==)
equal ⟨Char⟩ = (==)
equal ⟨Unit⟩ Unit Unit = True
equal ⟨Sum a b⟩ (Inl x) (Inl y)= equal⟨a⟩ x y
equal ⟨Sum a b⟩ (Inr x) (Inr y)= equal⟨b⟩ x y
equal ⟨Sum a b⟩ _ _ = False
equal ⟨Prod a b⟩ (x × y) (r × s)= equal⟨a⟩ x r ∧ equal⟨b⟩ y s
```

Figura 4.16: Definição de igualdade genérica.

Na Figura 4.16 é apresentada a definição da igualdade genérica em Generic Haskell. Na Figura 4.17 é definida a igualdade insensitiva de estruturas de dados que armazenam caracteres. Note que poderíamos definir equações para *somas* e *produtos* mas estas seriam idênticas às implementadas para a função `equal`. Então, ao invés de reimplementar todo esta funcionalidade em uma nova função genérica, especificamos que esta estende uma outra função utilizando a palavra chave **extends**. Neste exemplo, a linha:

```
ciequal extends equal
```

especifica que as equações definidas para `equal` devem ser *copiadas* para a nova função `ciequal`. Para este caso, somente a equação correspondente ao tipo `Char` não é copiada, pois `ciequal` realiza uma redefinição para este tipo.

```
ciequal ⟨a::⋆⟩ ::  a → a → Bool
ciequal extends equal
ciequal ⟨Char⟩ x y = (toUpper x) == (toUpper y)
```

Figura 4.17: Definição de igualdade usando Definições padrão.

4.3.8 Exemplos

Nesta seção serão apresentados mais exemplos de funções genéricas utilizando Generic Haskell. Por uma questão de coerência, serão mostrados os mesmos exemplos da Seção 4.2.1.

No primeiro exemplo, deseja-se construir uma função que realize o aumento do salário de todos os funcionários de uma empresa, que é representada por um tipo algébrico mutuamente recursivo. Uma abordagem natural para a solução deste problema é construir funções que utilizam *boilerplate code*. Alternativamente, pode-se construir uma função de ordem superior, que aplique a função que realiza o aumento de salário a um trecho específico do tipo de dados. Considerando esta segunda opção, podemos utilizar a função identidade genérica, mostrada na Figura 4.15. A partir desta definição, basta utilizar definições padrão, conforme pode ser visto no trecho de código a seguir:

```
update⟨a::⋆⟩::(update⟨a⟩)⇒ Float → a → a
update extends id
update⟨Salary⟩ v (S s) = S (v * (1 + s))
```

Nesta definição, a equação:

```
update⟨Salary⟩ v (S s) = S (v * (1 + s))
```

especifica o comportamento de `update` sobre o tipo `Salary`, realizando a operação de atualização. Observe que esta função usa a mesma estratégia utilizada na implementação com *SYB*, isto é, `update` realiza a travessia de toda a estrutura recursiva do valor do tipo `Company`, e comporta-se como a função identidade em todos os tipos diferentes de `Salary`.

Outro exemplo que utiliza o tipo de dados empresa, é a implementação de uma função que totaliza o valor gasto por esta empresa no pagamento do salário de seus funcionários. Esta função pode ser vista como um tipo de *fold* sobre o tipo algébrico empresa, operando apenas sobre valores de tipo `Salary` e ignorando os dos demais tipos. Para a definição desta função, será utilizada a função `gsum`, que realiza a soma de valores de interesse presentes em um tipo qualquer. O código para `gsum` é apresentado a seguir:

```
gsum⟨ a | c ⟩ ::  (gsum⟨ a | c ⟩, Num c) ⇒ a → c
gsum⟨Char⟩ x = 0
gsum⟨Int⟩ x = 0
gsum⟨Float⟩ x = 0
gsum⟨Unit⟩ x = 0
```

```
gsum⟨Sum a b⟩ (Inl x) = gsum⟨a⟩ x
gsum⟨Sum a b⟩ (Inr x) = gsum⟨b⟩ x
gsum⟨Prod a b⟩ (x × y) = (gsum⟨a⟩ x) + (gsum⟨b⟩ y)
```

Antes de comentar sobre a definição de `gsum`, é interessante analisar o tipo desta função. A anotação de tipo fornecida para esta função é:

```
gsum⟨ a | c ⟩ ::  (gsum⟨ a | c ⟩, Num c) ⇒ a → c
```

Nesta anotação, existem duas variáveis separadas por uma barra vertical '—'. Esta barra vertical, presente em anotações de tipo, é a maneira de especificar que um determinado subconjunto das variáveis de tipos presentes em uma anotação não são variáveis de parâmetros de tipo para a função que está sendo definida. Quaisquer variáveis que apareçam à direita de uma barra vertical são consideradas parametricamente polimórficas. No exemplo da definição de `gsum`, o tipo de retorno desta função é dado pela variável não-genérica `c`, que deve ser instanciada para um tipo que pertença à classe `Num`, conforme especificado na assinatura de `gsum`.

Analisando a definição de `gsum`, podemos perceber que esta função sempre retorna o valor zero para qualquer valor fornecido. Porém, podemos mudar seu comportamento, para que some termos presentes em um determinado ponto de um tipo algébrico qualquer. Sendo assim, a definição da função `salaryBill`, utilizando definições padrão, é:

```
salaryBill⟨a⟩ ::  (salaryBill⟨a⟩) ⇒ a → Float
salaryBill extends gsum
salaryBill⟨Salary⟩ (S s) = s
```

Esta função comporta-se de maneira idêntica a `gsum`, exceto quando esta é aplicada a valores do tipo `Salary`, caso em que retorna um valor de ponto flutuante correspondente ao salário de um determinado funcionário da empresa.

Parsers e *pretty-printers* são exemplos comuns de funções genéricas: converter um valor em um string legível ou converter um string para uma representação adequada para a manipulação em programas são tarefas presentes em diversos problemas.

Funções similares a `read` e `show` (ambas presentes nas bibliotecas da linguagem Haskell), necessitam acessar informações sobre construtores de dados, como seu nome e aridade, entre outras, para converter strings em valores de um tipo ou valores em strings. Para isto, Generic Haskell permite que padrões de tipo, em equações, assumam a forma:

`Con x` α

onde a variável `x` é um valor do tipo `ConDescr`. O tipo `ConDescr` é abstrato e possui diversas funções que podem ser utilizadas para obter informação sobre o construtor representado por um valor deste tipo. Dentre diversas operações suportadas, podemos citar:

- `conName :: ConDescr → String`: Esta função retorna um string correspondente ao nome do construtor representado por `ConDescr`.
- `conType :: ConDescr → String`: Esta função retorna um string correspondente ao nome do tipo ao qual este construtor pertence.
- `conArity :: ConDescr → Int`: Esta função retorna um número inteiro correspondente ao número de argumentos deste construtor.
- `conFixity :: ConDescr → Fixity`: Esta função retorna um valor que representa se um construtor é infixo ou não e seu respectivo valor de precedência. O tipo de dados Fixity é definido como:

```
data Fixity = NonFix | Infix {prec :: Int}
            | Infixl {prec :: Int} | Infixr {prec :: Int}
```

Exemplos reais da utilização de descritores de contrutores aparecem na implementação das funções `gread` e `gshow`, as equivalentes genéricas de `read` e `show`.

A função `gshow` é definida em termos de uma função auxiliar, `gshowP`, como pode ser visto no trecho de código seguinte:

```
gshow⟨a⟩::(gshowP⟨a⟩)⇒a→String
gshow⟨a⟩ = gshowP⟨a⟩ id
```

Todo o trabalho de conversão de um valor para o string que o representa é feito pela função `gshowP`, definida a seguir:

```
gshowP⟨a⟩::(gshowP⟨a⟩)⇒(String → String) → a→String
gshowP⟨Unit⟩ p Unit = ""
gshowP⟨Sum a b⟩ p (Inl x) = gshowP⟨a⟩ x
gshowP⟨Sum a b⟩ p (Inr x) = gshowP⟨b⟩ x
gshowP⟨Prod a b⟩ p (x × y) = (gshowP⟨a⟩ p x) ++ " " ++ (gshowP⟨b⟩ p x)
gshowP⟨Con c a⟩ p (Con x) = let parens x = "("++ x ++ ")"
                                body = gshowP⟨a⟩ parens x
                            in if null body then conName c
                            else p ((conName c) ++ " " ++ body)
gshowP⟨[a]⟩ p xs = let body = (concat
```

```
                        . instersperse ", "
                       .  map ⟨[ ]⟩((gshowP⟨a⟩ id)) xs
            in "[" ++ body ++ "]"
```

A função `gshowP` recebe um argumento a mais que `gshow`. Este argumento, de tipo String → String, é utilizado para a correta colocação de parênteses onde estes são necessários durante a construção do string.

A equação correspondente ao tipo estrutural `Unit` representa construtores que não possuem argumentos. Neste caso, o nome do construtor é o string de representação deste valor e este é gerado pela equação correspondente ao tipo `Con a`. Para valores cuja estrutura envolve um produto, é feita a concatenação dos resultados, utilizando como separador um caractere de espaço.

Grande parte do trabalho realizado por `gshowP` é feito pela equação correspondente ao tipo estrutural `Con`. Primeiramente, é construído um string correspondente aos argumentos deste construtor. Caso este construtor não possua argumentos, a variável `body` assume como valor um string vazio, fazendo com que o resultado seja o nome do construtor. Caso o construtor possua argumentos, estes são convertidos em strings e concatenados ao nome do construtor, formando um string que será envolvido por parênteses.

A última equação faz a conversão de valores de tipo lista utilizando a sintaxe de Haskell para listas.

Evidentemente que tipos primitivos como `Int`, `Float` e `Char` devem implementar esta operação utilizando alguma função primitiva.

A definição de uma função genérica para realizar o *parsing* de um string para seu correspondente valor é similar a definição de `gshow`. A diferença básica entre `gread` e `gshow` é que a primeira deve lidar com a entrada parcialmente consumida e com a ocorrência de falhas. O código de `gread` é mostrado a seguir:

```
gread⟨t::⋆⟩ ::  (greadsPrec⟨t⟩) ⇒ String → t
gread⟨t⟩ s = case [x | (x, s1) ← greadsPrec⟨t⟩ ConStd 0 s] of
                [y] → y
                [ ] → error "erro de parsing."
                  _ → error "gread:  ambigüidade."
```

A definição de `gread` utiliza uma função auxiliar `greadsPrec`, para construir uma lista de possíveis resultados do *parsing* de um string. Evidentemente, o processo de *parsing* de um string deve produzir um único resultado, que é tratado pela primeira equação pertencente à expressão `case` presente em `gread`. Caso seja retornada uma lista com mais de um elemento, ou uma lista vazia, significa que uma ambigüidade foi encontrada

(mais de um valor produzido a partir de um mesmo string), ou foi detectado um erro de *parsing*, respectivamente.

A função `greadsPrec` realiza o *parsing* de um string fornecido como parâmetro. Esta função recebe como primeiro argumento um valor do tipo `ConWhat`, que indica se um determinado construtor é infixo ou não. Valores deste tipo são utilizados, durante o processo de *parsing*, para realizar o correto processamento de separadores presentes em um string. A definição do tipo `ConWhat` é mostrada a seguir:

```
data ConWhat = ConStd | ConIfx String
```

O construtor `ConStd` representa construtores não infixos e `ConIfx` construtores infixos. Estes valores são utilizados em valores auxiliares, que identificam os tipos de separadores que são utilizados em strings para construtores infixos e não infixos. Esta tarefa é implementada pela função `separator`:

```
separator ::  ConWhat → String → [(Char, String)]
separator ConStd s = [(' ', s)]
separator (ConIfx cname) s = [(',', s1)| (cname', s1) ← lex s,
                                       | cname == cname']
```

A função `separator` retorna uma lista de pares formados por um caractere separador e um string. Para construtores não infixos, o caractere separador é um espaço, que é retornado em um par, junto com o nome do construtor. Para construtores infixos, é feita a remoção do nome deste construtor do string, e o restante deste é retornado como resultado, em um par cujo primeiro componente (caractere separador) é uma vírgula. Esta função é utilizada na equação que define como realizar o *parsing* para tipos cuja estrutura é `Prod a b`. Nesta equação, `separator` realiza a remoção de separadores que possam estar contidos entre o primeiro componente (que já foi processado) e o segundo componente a ser processado.

A definição de `greadsPrec` realiza a análise de casos sobre o tipo do valor a ser retornado como resultado. A primeira equação desta definição é feita sobre o tipo `Unit`, que representa construtores sem parâmetros. Para este caso é retornada uma lista, contendo um par formado por um valor de tipo `Unit` e o restante da string. Para tipos cuja estrutura envolve uma soma, são montadas duas listas: uma correspondente aos valores de tipo `a -` que devem ser construídos usando `Inl`, e uma correspondente aos valores de tipo `b -` que devem ser construídos usando `Inr`. A equação que define comportamento de `greadsPrec` para tipos produto constrói uma lista de valores `(a` $\times$ `b)`, a partir da string fornecida como parâmetro. O primeiro componente deste produto é obtido fazendo uma chamada a `greadsPrec`$\langle$`a`$\rangle$, que retorna como resultado um par, contendo um valor de tipo `a` e o restante do string a ser consumido. O restante do

string é passado para a função `separator`, para a remoção de possíveis separadores existentes entre os valores que compõe o produto. Após a remoção de separadores, o restante do string é passado para a chamada `greadsPrec⟨b⟩`, que retorna um par composto por um valor de tipo `b` e um string que ainda deve ser processado. A equação correspondente a `Con a`, realiza o *parsing* de um construtor de acordo com o tipo de construtor especificado por um valor do tipo `ConWhat`, realizando o tratamento para construtores infixos e não infixos. Nesta definição de `greadsPrec` foram omitidas equações para tipos primitivos, que devem fornecer alguma função para realizar a conversão de strings para valores destes tipos. A seguir é mostrada a definição de `greadsPrec`:

```
greadsPrec⟨a::⋆⟩::(greadsPrec⟨a⟩)⇒ConWhat→Int→ String→[(a, String)]
greadsPrec⟨Unit⟩ f p s = [(Unit, s)]
greadsPrec⟨Sum a b⟩ f p s
          = map (λ (a, s)→ (Inl a, s)) (greadsPrec⟨a⟩ f p s) ++
            map (λ (b,s) → (Inr b, s)) (greadsPrec⟨b⟩ f p s)
greadsPrec⟨Prod a b⟩ f p s
             = [(a × b, s3) |
               , (a, s1 )← (greadsPrec⟨a⟩ f p) s
               , (_, s2) ← separator f (stripWhite s1)
               , (b, s3) ← (greadsPrec⟨b⟩ f p) (stripWhite s2)
greadsPrec⟨Con c a⟩ f p s
       = case (conFixity c) of
                 (_, NonFix) →
                     readParen (p > (getFixity c) && (conArity c) > 0)
                         (λ s) → [(Con x, s2)
                             | (t, s1) ← lex s
                             , t == conName c
                             , (x, s2) ← greadsPrec⟨a⟩ ConStd 10 s1]) s
                          _ →
                               readParen (p > (getFixity c) && (conArity c) > 0)
                                (λ s → [(Con x, s2)
                                | (x, s2) ← greadsPrec⟨a⟩(ConIfx(conName
                                      c)) (getFixity c + 1) s]) s
```

4.4 Análise Comparativa

Esta seção apresenta uma análise comparativa entre duas abordagens para programação genérica em Haskell: *Scrap you boilerplate* (SYB) (apresentada na Seção 4.2) e *Generic Haskell* (GH) (apresentada na Seção 4.3).

Como vimos nas seções anteriores, as principais características das abordagens SYB são:

- Provê uma biblioteca de funções genéricas para travessia de valores de tipos de complexos (`foldT`, `gmapT` etc)
- Uma função genérica é definida em termos de funções sobrecarregadas, definidas em instâncias da classe `Data`. Instâncias da classe `Data`, por sua vez, apenas podem ser definidas para tipos representáveis, ou seja, que são instâncias da classe `Typeable`.
- Funções genéricas têm como parâmetro funções polimórficas, ou seja, SYB requer polimorfismo de ordem superior.
- Permite a definição de funções polimórficas que operam de modo específico para valores de determinados tipos, por meio da utilização de operadores de extensão de funções: `mkT` e `mkQ`. Esses operadores são definidos em termos de uma função de coersão de tipos segura – `cast` –, cuja definição usa a função `typeOf`, que deve ser definida para cada novo construtor de tipos, como instância da classe `Typeable`.
- Funções genéricas podem ser derivadas automaticamente, para cada tipo definido pelo programador, por meio do mecanismo de `deriving`.

As principais características da abordagem GH são:

- Estende a linguagem Haskell, adicionando construções que permitem a definição de funções indexadas por tipos e tipos indexados por tipos (λ-calculus de ordem superior).
- Utiliza esse mecanismo de funções indexadas por tipos e a definição de um tipo de representação estrutural de tipos, para possibilitar a definição de funções genéricas.
- Para cada aplicação de uma função genérica a valores de um determinado tipo, é automaticamente gerada uma instância dessa função genérica para este tipo. A geração dessa instância é baseada em funções de isomorfismo entre o tipo

e sua representação estrutural, sendo as funções que definem esse isomorfismo também geradas automaticamente, para cada tipo definido. Instâncias de funções genéricas, assim como de funções de isomorfismo, são, portanto, funções sobrecarregadas.

- Possibilita a definição de funções polimórficas especializadas (no nível da linguagem GH e não de Haskell), possibilitando especialização local.

As duas abordagens atingem os principais objetivos da programação genérica, ou seja, possibilitar a definição de funções que operam sobre valores de diferentes tipos, que se adaptem automaticamente a mudanças na definição da estrutura desses tipos, e que possam ser usadas para definir novas funções sobre esses tipos, evitando, a necessidade de *boilerplateplate code* em programas.

A análise comparativa a seguir é guiada pelos exemplos de funções genéricas apresentados anteriormente, e utiliza alguns critérios apresentados em [HJL06] e outros definidos neste trabalho. Uma explicação breve sobre cada um dos critérios utilizados será dada na próxima seção.

4.4.1 Critérios para Comparação de Abordagens de Programação Genérica

4.4.1.1 Princípio de Completude de Tipos

O princípio de completude de tipos (*type completeness principle* [Wat90]) afirma que nenhuma operação de uma linguagem de programação deve ser arbitrariamente restrita sobre os tipos de seus operandos. Por exemplo, em Haskell uma função pode receber um argumento de qualquer tipo, inclusive uma função, e tuplas ou tipos algébricos arbitrários podem ser formados por funções. Por isto dizemos que Haskell satisfaz o princípio de completude de tipos no nível de valores. Evidentemente existem exceções. Por exemplo, não é possível em Haskell que uma função polimórfica seja passada como argumento (todavia existem extensões que permitem este recurso conhecido como tipos de *rank superior*). No entanto, tais restrições não são em geral caracterizadas como arbitrárias. No caso de argumentos que são funções de *rank-superior*, por exemplo, a inferência de tipos se torna em geral indecidível [PVWS07].

A partir do princípio de completude de tipos pode-se definir o seguinte critério:

Reflexividade: Uma linguagem de programação genérica é dita ser *reflexiva* se uma função genérica pode ser usada sobre valores de quaisquer tipos que possam ser representados nesta linguagem.

4.4.1.2 Sistema de Tipos

Sistemas de tipos de linguagens de programação são úteis para evitar a ocorrência de erros em tempo de execução que podem levar a interrupção da execução de programas. A mais importante característica de uma linguagem, em relação a seu sistema de tipos, é a segurança, definida abaixo:

Segurança (Type safety): Uma linguagem de programação é dita ser segura (*type safe*) se não há a possibilidade de interrupção da execução de um programa devido a um erro de tipo não detectado durante o processo de verificação[HJL06].

4.4.1.3 Informações Sobre Tipos

Este critério está relacionado à expressividade da linguagem de tipos e o que podemos inferir, sobre uma função genérica, a partir de seu tipo.

Clareza de tipos de funções genéricas: Este critério expressa o que podemos dizer sobre o tipo de uma função genérica. Tipos podem ser utilizados para inferir propriedades úteis, como *teoremas gratuitos (free theorems)* [Wad89]. Os tipos inferidos ou declarados para funções genéricas correspondem ao tipo que intuitivamente atribuiríamos a esta função? Por exemplo, uma função de tipo $a \rightarrow a \rightarrow Bool$ é provavelmente uma função de comparação e uma função de tipo $(a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$ é um *map* para algum construtor de tipos f.

4.4.1.4 Engenharia de Software

Este critério diz respeito a uma das questões mais relevantes de engenharia de software: extensibilidade. Ou seja, a linguagem permite a definição de funções genéricas extensíveis?

Extensibilidade: Diz-se que uma função genérica é extensível (ou aberta) se esta permite a inclusão de definições específicas para casos particulares em diferentes pontos do programa. Extensibilidade é útil por permitir o desenvolvimento incremental de software, em que novas funcionalidades são adicionadas à medida em que o projeto de software evolui. Aparentemente, este critério de comparação pode parecer contraditório em relação ao conceito de programação genérica, onde uma função possui o comportamento determinado pela estrutura do tipo ao qual esta é aplicada. Porém, em diversas situações é necessário adicionar novas definições para casos específicos de funções genéricas para prover um comportamento personalizado para um determinado tipo. Como exemplo, considere a tarefa de desenvolver uma função genérica para realizar o *pretty-printing* de um determinado valor. Para determinados tipos, o string gerado por um pretty-printing genérico pode não ser adequado, pois o desenvolvedor pode desejar que este string tenha um formato específico, que atenda suas necessi-

dades, seja produzido pela função ao invés de concatenar strings correspondentes a construtores de dados ao resultado de converter cada um de seus argumentos.

4.4.1.5 Projeto da Linguagem

A tarefa de projeto de uma linguagem de programação é difícil devido ao grande número de conceitos que podem ser incluídos ou não em uma linguagem [Wat90]. Todas as abordagens de programação genérica devem fornecer aproximadamente os mesmos conceitos: polimorfismo paramétrico, polimorfismo de sobrecarga e uma representação estrutural de tipos. Porém, consideraremos como critério de avaliação o que chamaremos de *simplicidade*, que será definida a seguir:

Simplicidade: Este critério procurará medir o quanto é simples desenvolver uma determinada função genérica na linguagem avaliada. Entenda-se por *simplicidade* o quão intuitiva é a abordagem de programação genérica considerada. Há a necessidade do programador conhecer detalhes de como a inferência de tipos opera? Ele deve estar habituado com conceitos teóricos de linguagens de programação para implementar funções genéricas?

4.4.2 Análise das Abordagens de Programação Genérica

Nesta seção é apresentado o resultado da nossa análise das abordagens para programação genérica SYB e Generic Haskell, segundo os critérios apresentados na seção anterior. Para guiar esta comparação, são utilizados os exemplos apresentados anteriormente, ou seja, as funções `salaryBill` e `increaseSalary`, definidas sobre o tipo de dado algébrico `Company`, além de definições genéricas para igualdade e *pretty-printing*.

4.4.2.1 Scrap your Boilerplate

Princípio de Completude de Tipos

Reflexividade: SYB não constitui uma abordagem totalmente reflexiva, uma vez que a definição de funções genéricas está atrelada à definição de uma instância da classe `Typeable`, mas instâncias desta classe só podem ser definidas para tipos algébricos cujas variáveis de tipo são de *kind* $\star$.

Sistema de Tipos

Segurança (Type safety): A abordagem SYB não é totalmente segura. A segurança pode ser violada por um mau uso do mecanismo de extensão de tipos, por meio dos operadores `mkT` e `mkQ`. O problema é que esses operadores são definidos em termos de um operador de coerção segura de tipos, que

por sua vez é definido em termos do operador `typeOf`, definido na classe `Typeable`. O programador pode definir instâncias dessa classe que podem causar erros de tipo em tempo de execução, devido a conversões indevidas de tipos.

Informações Sobre Tipos

Clareza de tipos de funções genéricas: Tipos de funções genéricas implementadas usando SYB correspondem aos tipos que intuitivamente deveriam ser atribuídos a essas funções, a menos de uma ou mais restrições de classes do SYB presentes no contexto do tipo da função genérica.

Engenharia de Software

Extensibilidade: Em [LP05] é apresentada uma abordagem alternativa para a implementação de funções genéricas no SYB que são *extensíveis*. A principal idéia desta abordagem é utilizar o mecanismo de classes / instâncias de Haskell para permitir a definição de funções genéricas extensíveis. Apesar de já existir uma publicação relativa a esta abordagem, sua implementação ainda não está presente nas bibliotecas que acompanham o compilador GHC.

Projeto da Linguagem

Simplicidade: A biblioteca SYB possui diversos combinadores que permitem a definição de funções genéricas de maneira simples e intuitiva. Porém, caso o programador necessite de algum combinador que não esteja presente na biblioteca, ou que não possa ser expresso em termos de combinadores de mais alto nível desta biblioteca, o processo de definição de novos combinadores em termos de combinadores mais básicos pode não ser muito simples, devido aos diversos conceitos utilizados por esta biblioteca de combinadores.

4.4.2.2 Generic Haskell

Princípio de Completude de Tipos

Reflexividade: Generic Haskell é totalmente reflexiva em relação a tipos que podem ser definidos dentro do padrão Haskell 98, exceto por restrições presentes em definições de tipos de dados. Em Haskell, é possível especificar restrições sobre variáveis de tipos, presentes em definições de tipos de dados. Como exemplo, considere (exemplo retirado de [Jon03]):

```
data Eq a ⇒ Set a = NilSet | ConsSet a (Set a)
```

A restrição `Eq a` garante que o construtor `ConsSet :: a → Set a → Set a` apenas seja aplicado a um valor de um tipo que seja instância da classe `Eq`.

Sistema de Tipos

Segurança (Type safety): O sistema de tipos de Generic Haskell é descrito em [Löh04]. Entretanto, a versão mais recente do compilador desta linguagem ainda não imlementa este sistema de tipos. Como Generic Haskell realiza a tradução para código Haskell, de tipos indexados por tipos e de funções indexadas por tipos, a verificação de tipos pode ser deixada para um compilador de Haskell, como o GHC. Como Haskell é uma linguagem segura, podemos afirmar que Generic Haskell também é.

Informações Sobre Tipos

Clareza de tipos de funções genéricas: Os tipos de funções genéricas, em Generic Haskell, são próximos dos tipos que esperaríamos para esta função, exceto pela presença de dependências. Como exemplo, considere o tipo da igualdade genérica (apresentada na Figura 4.16):

$$\texttt{equal}\ \langle \texttt{a::}\star \rangle\ \texttt{::}\ \ \texttt{(equal}\langle\texttt{a}\rangle\texttt{)} \Rightarrow \texttt{a} \rightarrow \texttt{a} \rightarrow \texttt{Bool}$$

Embora o tipo atribuído à da função `equal` seja bastante natural, isso não ocorre para as funções genéricas. Como exemplo, considere o tipo de `map` em Generic Haskell, cuja definição será apresentada no Apêndice A:

$$\texttt{map}\langle \texttt{a ::}\star\texttt{, b::}\star \rangle\ \texttt{::}\ \ \texttt{(map}\langle\texttt{a, b}\rangle\texttt{)} \Rightarrow \texttt{a} \rightarrow \texttt{b}$$

Observe que este tipo não corresponde ao tipo que intuitivamente se atribuiria a `map`. Isto se deve ao processo de especialização de tipos, conhecido como aplicação genérica, utilizado pelo compilador de Generic Haskell. Por estar fora do escopo deste trabalho, este processo de especialização foi omitido. O leitor interessado pode recorrer ao Capítulo 6 de [Löh04], que apresenta em detalhes este processo.

Engenharia de Software

Extensibilidade: Generic Haskell, por meio do recurso de definições padrão (Seção 4.3.7), permite que novos casos sejam adicionados a uma definição de função genérica. Porém, a utilização deste recurso não define funções genéricas extensíveis, uma vez que o compilador de Generic Haskell copia todos os casos, presentes na definição original, para a nova função que utiliza definições padrão.

Projeto da Linguagem

Simplicidade: Generic Haskell é a mais poderosa abordagem existente para programação genérica em Haskell. Essa abordagem permite a definição de tipos indexados por tipos, funções indexadas por tipos, e estes tipos podem envolver construtores de tipos de *kind* arbitrário. Generic Haskell utiliza uma representação estrutural baseada em somas de produtos (Seção 4.1.1), que facilita definir funções genéricas que não fazem uso direto de operadores de recursão como, por exemplo, cata- e anamorfismos genéricos. Em [Löh04] são apresentadas definições de cata- e anamorfimos genéricos, a partir da representação de tipos de dados por meio de pontos fixos de um functor regular. Esta abordagem é similar à utilizada pela linguagem PolyP [JJ97], para a definição de funções genéricas para tipos de dados regulares.

Além disso, o mecanismo de especialização de tipos de funções genéricas, utilizado por Generic Haskell, deve ser compreendido pelo programador para a definição de algumas funções, como por exemplo, a função *map* genérica.

Devido à necessidade do conhecimento de mecanismos utilizados pelo compilador e de diferentes maneiras para representar estruturalmente funções, pode-se concluir que a utilização de Generic Haskell, para o desenvolvimento de alguns tipos de funções, não atende completamente ao critério de simplicidade, exigindo do programador profundo conhecimento dos conceitos em que se baseia a linguagem.

4.4.2.3 Conclusões

A Figura 4.18 mostra os resultados da avaliação das duas abordagens para programação genérica consideradas neste capítulo. Nesta figura, a notação n/m indica que n dos m itens de um determinado critério foram atendidos. Evidentemente, o objetivo desta análise não é afirmar qual das três abordagens é a melhor para o desenvolvimento de funções genéricas em Haskell, mas sim avaliar possíveis limitações presentes em cada uma destas abordagens. Esperamos que esta avaliação seja utilizada como um direcionamento para possíveis melhoramentos em cada uma dessas abordagens.

	Completude	Sist. de Tipos	Infor.	Eng. Software	Projeto
SYB	0/1	0/1	1/1	1/1	1/1
GH	1/1	1/1	1/1	1/1	0/1

Figura 4.18: Avaliação de resultados da análise comparativa entre SYB e Generic Haskell.

A abordagem SYB não atende satisfatoriamente os critérios de segurança de tipos e completude. A segurança de tipos pode ser violada por meio de coersão de tipos indevida. O critério de completude não é satisfeito, uma vez que, nesta abordagem, não é possível escrever funções genéricas para quaisquer tipos da linguagem Haskell.

Por sua vez, Generic Haskell não atende o critério de projeto, uma vez que esta abordagem exige do programador conhecimentos mais especializados, como a representação estrutural de tipos e a definição de funções e tipos algébricos parametrizados por tipos. A definição de funções genéricas nesta abordagem é, em geral, mais difícil que em SYB.

4.4.2.4 Considerações Finais

Uma pergunta que surge a partir da análise apresentada neste capítulo é: quando se deve utilizar uma determinada abordagem ou outra? Nesta seção, procuramos responder esta pergunta.

A biblioteca SYB é mais adequada para aplicações que lidam com tipos de dados complexos, quando se deseja realizar alguma operação em pontos específicos desse tipo. Nesse caso, a utilização das funções genéricas disponíveis na biblioteca SYB facilita o desenvolvimento do programa e resulta em um código conciso e legível. Esse código é automaticamente adaptável a posteriores mudanças na definição da estrutura do tipo de dados, já que funções genéricas são derivadas automaticamente para a nova definição do tipo.

Como mencionamos anteriormente, a codificação de algumas funções genéricas pode requerer o uso de combinadores de mais baixo nível dessa biblioteca. Nesse caso, a definição dessas funções demanda um profundo conhecimento do programador sobre os conceitos e a estrutura de classes e instâncias dessa biblioteca, tornando a sua programação mais difícil. Além disso, essa estrutura de classes e instâncias constitui uma restrição para que esta abordagem possa ser facilmente estendida.

Generic Haskell é a abordagem mais completa e expressiva para programação genérica em Haskell, dentre as atualmente disponíveis, podendo ser usada para definir um conjunto maior de funções genéricas. Entretanto, a programação de funções genéricas nesta abordagem é mais difícil, pois envolve conceitos aos quais o programador em geral não está habituado, como a parametrização de definições de funções e de definições de tipos de dados por tipos. Além disso, um problema inerente à abordagem utilizada por Generic Haskell é a possível ineficiência causada pelo número excessivo de conversões para a representação intermediária de tipos utilizada pela linguagem.

Capítulo 5

Programação Genérica no CT

Este capítulo discute como prover suporte a programação genérica no sistema CT, evitando que a definição de funções em uma linguagem baseada neste sistema possa envolver código repetitivo, que apenas realiza uma travessia sobre uma estrutura de dados, sem realizar nenhuma operação útil. São identificadas extensões ao sistema CT necessárias para que isso possa ser feito, sem necessidade de definir funções baseadas em indução estrutural tal como em Generic Haskell, e sem utilizar um mecanismo de coersão de tipos potencialmente inseguro tal como em SYB.

A primeira seção deste capítulo mostra como exemplos apresentados anteriormente podem ser escritos em uma linguagem baseada no sistema CT, utilizando definições de funções sobrecarregadas sobre construtores de tipos e definições sobrecarregadas com tipos possivelmente sobrepostos[1]. Embora seja possível definir funções *extensíveis* facilmente no sistema CT, usando sobrecarga, isso não é suficiente para suporte a programação genérica, pois a defnição de funções sobre tipos complexos ainda envolveria código *boilerplate*.

Com base no exame desses exemplos, buscamos identificar as extensões fundamentais que devem ser incorporadas ao sistema CT para prover um suporte adequado a programação genérica, sendo essas apresentadas nas seções posteriores.

5.1 Definição de funções usando sobrecarga e instâncias sobrepostas

Em sua definição original, o sistema de tipos CT não permite a definição de instâncias sobrepostas [Vas04]. Antes de definir uma extensão ao sistema CT para suporte a definição de instâncias sobrepostas, apresentamos uma motivação para a sua utilização.

[1] Similares a definições permitidas em implementações de Haskell conhecidas como *overlapping instances*.

Considere, novamente, a tarefa de implementar a função `increaseSalary`, que aumenta o salário de todos os funcionários de uma empresa, contidos em um valor de um tipo algébrico que representa a estrutura organizacional desta empresa, conforme apresentado na Figura 4.9. Uma das idéias utilizadas na abordagem do SYB é estender o comportamento de uma função que opera sobre um único tipo t, para uma que opera sobre diversos tipos, comportando-se como a função identidade sobre todos os tipos, exceto t. Para isto, SYB utiliza o operador de extensão `mkT` (definido na Figura 4.4), o qual é definido em termos da função de coerção de tipos `unsafeCoerce`. Um comportamento similar pode ser obtido utilizando definições sobrecarregadas com tipos sobrepostos, ou seja, unificáveis, conforme mostrado a seguir. Suponha que, em um contexto Γ_{incS}, existam as seguintes definições do símbolo `incS`:

```
overload incS ::  Float → Salary → Salary
         incS k (S s) = S(s * (1 + k))
overload incS _ v = v
```

As duas definições de `incS` possuem os tipos mostrados na Figura 5.1. A primeira definição define um comportamento específico da função para valores do tipo `Salary`. Para estender essa função para valores de outro tipos, de maneira similar ao que se obtém por meio do operador `mkT` de SYB, basta adicionar uma segunda definição deste símbolo, de comportamento igual ao da função identidade.

```
incS ::  Float  →  Salary  →  Salary
incS ::  a      →  b       →  b
```

Figura 5.1: Tipos da função incS

Os tipos das duas definições de `incS` são unificáveis, por meio da substituição:

$$S = \{a \mapsto Float, b \mapsto Salary\}$$

O tipo inferido para `incS` em um contexto Γ_{ins}, que contenha os tipos das definições sobrecarregadas acima, é:

$$\texttt{incS} \ : \ \forall\alpha,\beta.\{\texttt{incS}{:}\alpha \to \beta \to \beta\}.\texttt{incS}{:}\alpha \to \beta \to \beta$$

Considere, agora, as seguintes expressões, que utilizam `incS`:

```
(incS 1.5) True
(incS 1.5) (S 100)
```

As avaliações destas expressões produzem os resultados, `True` e `150`, respectivamente. Quando a expressão `incS 1.5` é aplicada a um valor do tipo `Bool`, o tipo inferido tem uma restrição que informa que deve existir, neste contexto, uma definição de `incS` que possua um tipo que case com o tipo requerido para `incS` nesta expressão (o tipo requerido, para `incS`, nesta expressão é `Float` $\rightarrow$ `Bool` $\rightarrow \alpha$). A definição de `incS` que possui o tipo $\alpha \rightarrow \beta \rightarrow \beta$ é a única, no contexto Γ_{incS}, que casa com a restrição `incS:Float` $\rightarrow$ `Bool` $\rightarrow \alpha$. Como esta definição comporta-se como a função identidade, a expressão `(incS 1.5) True` é avaliada como `True`. Por sua vez, a expressão `(incS 1.5) (S 100)` possui a restrição `incS : Float` $\rightarrow$ `Salary` $\rightarrow$ `Salary`, que casa com a definição de `incS` que possui o tipo `incS : Float` $\rightarrow$ `Salary` $\rightarrow$ `Salary`. Portanto, a expressão `(incS 1.5) (S 100)` é avaliada para `S 150`.

A próxima seção apresenta, informalmente, uma extensão do sistema CT para que este permita definições sobrepostas de símbolos sobrecarregados.

5.1.1 Definições Sobrepostas no Sistema CT

A utilização de definições sobrecarregadas sobrepostas, ou seja, cujos tipos são unificáveis, é útil para a construção de alguns programas, como ilustra o exemplo anterior. Nesta seção, apresentamos como a política de sobrecarga do sistema CT pode ser modificada de modo a permitir tais definições.

Antes disso, apresentamos, informalmente, a abordagem utilizada pelo compilador Haskell GHC para verificar a validade de um conjunto de definições de instâncias sobrepostas.

5.1.1.1 Tratamento de instâncias com tipos sobrepostos no GHC

O comportamento padrão do compilador GHC é não permitir instâncias com tipos sobrepostos. Entretanto, esse comportamento pode ser alterado por meio da opção de compilação `-fallow-overlapping-instances`. Nesse caso, definições de instâncias sobrepostas são permitidas, desde que, para qualquer restrição de classe de tipo, seja possível determinar, de maneira unívoca, uma instância mais específica que satisfaça essa restrição. Caso não exista uma única instância mais específica que satisfaça a restrição, o programa é rejeitado.

Para exemplificar como GHC realiza a verificação da validade de um conjunto de instâncias sobrepostas, considere o trecho de código apresentado na Figura 5.2.

Suponha que, em um dado ponto do processo de inferência / verificação de tipo, o compilador GHC tente resolver a seguinte restrição `C Int [Int]`. Evidentemente, neste contexto, existem duas intâncias que satisfazem esta restrição: as instâncias 3 (`C Int [a]`) e 4 (`C Int [Int]`). Porém, como o tipo da instância 4 é mais específico

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

1. `instance C Int a where ...`
2. `instance C a Bool where ...`
3. `instance C Int [a] where ...`
4. `instance C Int [Int] where ...`

Figura 5.2: Exemplo de sobreposição inválida de tipos de instâncias.

que o tipo da instância 3 em relação à restrição `C Int [Int]`, este será escolhido pelo compilador GHC. Suponha, agora, que GHC deseje resolver a restrição `C Int Bool`. Observando o conjunto de instâncias apresentado na Figura 5.2, existem duas possíveis instâncias que podem satisfazer esta restrição: instâncias 1 e 2. Como pode ser observado, neste caso não é possível determinar, para a restrição `C Int Bool`, uma instância que tenha um tipo mais específico que satisfaça a restrição, uma vez que nenhum tipo dentre `C Int a` e `C a Bool` é mais específico que o outro. Nos casos em que isso ocorre, o GHC rejeita o programa.

5.1.1.2 Definições com Tipos Sobrepostos no Sistema CT

A aceitação de definições com tipos sobrepostos no sistema CT pode ser formalizada pela seguinte regra:

Suporte a definições com tipos sobrepostos no Sistema CT:

Seja Γ um contexto de tipos tal que $X \subseteq \Gamma$, onde $X = \{x : \sigma_1, \ldots, x : \sigma_n\}$ corresponde a um conjunto de definições possivelmente sobrepostas de um símbolo x e $\sigma_i = \forall\overline{\alpha_i}.\kappa_i.\tau_i$ é o tipo, com restrições, para a i-ésima definição de x. Diz-se que Γ é um contexto válido se, e somente se, para qualquer conjunto de definições sobrepostas de um símbolo x, a seguinte condição é satisfeita:

$$\exists j \in \{1 \ldots n\}.\forall i \in \{1 \ldots n\}.i \neq j \rightarrow \tau_i \neq \tau_j \wedge (unify(\tau_i, \tau_j, S) \rightarrow match(\tau_i, \tau_j, S)$$

As relações $unify(\tau_i, \tau_j, S)$ e $match(\tau_i, \tau_j, S)$ são definidas a seguir:

$$unify(\tau_i, \tau_j, S) = S\tau_i = S\tau_j$$

$$match(\tau_i, \tau_j, S) = S\tau_i = \tau_j$$

A definição de aceitação de instâncias sobrepostas no sistema CT mostra que um conjunto X, de tipos possivelmente sobrepostos, para um determinado símbolo sobre-

carregado, somente é válido se contém um tipo τ que é mais específico do que todos os demais, ou seja, $\forall \tau' \in X. \exists S. S\tau' = \tau$.

Como exemplo, considere o trecho de código Haskell mostrado na Figura 5.2. Neste trecho, temos que o símbolo sobrecarregado `c` possui os tipos apresentados na Figura 5.3:

```
c :: Int  → a
c :: a    → Bool
c :: Int  → [a]
c :: Int  → [Int]
```

Figura 5.3: Tipos de definições sobrepostas para o símbolo `c`.

Observe que não é possível determinar um tipo mais específico para este conjunto de tipos do símbolo `c`, uma vez que os tipos `a` → `Bool` e `Int` → `a` são unificáveis, mas não é possível determinar uma substituição S que torna um deles igual ao outro.

Como um último exemplo da utilização desta política de permissão de instâncias sobrepostas, considere a definição da função `incS`, cujos tipos foram apresentados na Figura 5.1. Como os tipos das duas definições de `incS` são unificáveis, deve existir uma substituição que instancia um deles para o outro. É fácil perceber que a substituição:

$S_1 = \{a \mapsto$ `Float`, $b \mapsto$ `Salary` $\}$

torna o tipo `a` → `b` → `b` igual ao tipo `Float` → `Salary` → `Salary`. Portanto, este conjunto de instâncias sobrepostas é válido.

5.2 Extensões ao CT para suporte a Programação Genérica

Esta seção apresenta a codificação, em uma linguagem baseada no sistema CT, de 3 dos exemplos utilizados anteriormente para ilustrar as abordagens SYB e Generic Haskell: `increaseSalary`, `salaryBill` e igualdade. Primeiramente apresentamos uma codificação destas funções utilizando apenas os recursos atuais do sistema CT, para mostrar como tais definições consistem de puro código boilerplate e não é suficientemente genérica para adaptar-se a modificações na estrutura dos tipos de seus parâmetros.

Com base nesses exemplos, avaliamos as limitações atuais do sistema CT e propomos extensões que visam solucionar os problemas encontrados. Finalizamos esta seção apresentando uma nova codificação desses três exemplos, em uma linguagem baseada

no sistema CT com as extensões propostas, e comparamos esta abordagem com Generic Haskell e *Scrap your boilerplate*.

Considere, primeiramente, a implementação de uma função que aumenta os salários de todos os funcionários de uma empresa, representada pelo tipo `Company` (definido na Figura 4.9). A definição desta função, a seguir, utiliza definições sobrecarregadas para cada um dos tipos que compõem o tipo `Company`:

```
overload increase v (C ds) = C (map (increase v) ds);

overload increase v (D n e s) = D n (increase v e)(map(increase v) s);

overload increase v (PU e) = PU (increase v e);
         increase v (DU s) = DU (increase v s);

overload increase v (E p s) = E p (increase v s);

overload increase v (S s) = S ((1.0 + v) * s);
```

Observe que a definição de `increase`, apesar de simples e direta, consiste quase exclusivamente de código *boilerplate*. Esta função percorre recursivamente a estrutura de cada um dos tipos e, ao encontrar um elemento do tipo `Salary`, realiza a alteração de seu valor, de acordo com a porcentagem fornecida como primeiro argumento.

A definição da função `salaryBill`, que totaliza o valor de salário de todos os funcionários da empresa, faz uso da função `collect`, que percorre a estrutura do tipo `Company`, coletando os valores de salário em uma lista. Note que a definição da função `collect` também consiste em puro código *boilerplate*.

```
overload collect x = [];
overload collect (S s) = [s];
overload collect (C ds) = concatMap collect ds;
overload collect (D n e s) = (concatMap collect n) ++
                             (collect e) ++
                             (concatMap collect s);
overload collect (PU e) = collect e;
         collect (DU s) = collect s;
overload collect (E p s) = (collect p) ++ (collect s);
overload collect (P n a) = (concatMap collect n) ++
                           (concatMap collect a);
```

```
salaryBill = sum . collect;
```

Um exemplo clássico de definição de função genérica é a comparação de valores quanto a igualdade. A definição de igualdade para um determinado tipo algébrico é, em grande parte, puro código *boilerplate*. O trecho de código seguinte ilustra a definição da igualdade para os tipos algébricos que constituem o tipo `Company`:

```
overload (==) = eqInt;

overload (==) x y = (ord x) == (ord y);

overload (==) = eqFloat;

overload (==) [] [] = True;
         (==) (x:xs) (y:ys) = (x == y) && (xs == ys);
         (==) _ _ = False;

overload (==) (C d1) (C d2) = d1 == d2;

overload (==) (D n1 e1 s1) (D n2 e2 s2) = (n1 == n2) &&
                                          (e1 == e2) &&
                                          (s1 == s2);

overload (==) (PU e1) (PU e2) = e1 == e2;
         (==) (DU d1) (DU d2) = d1 == d2;
         (==) _ _ = False;

overload (==) (E p1 s1) (E p2 s2) = (p1 == p2) && (s1 == s2);

overload (==) (P n1 s1) (P n2 s2) = (n1 == n2) && (s1 == s2);

overload (==) (S f1) (S f2) = f1 == f2;
```

Figura 5.4: Igualdade definida usando o Sistema CT

A definição de igualdade apresentadas na Figura 5.4 é também simples e direta. Para estender essa definição para um novo tipo, basta adicionar definições sobrecarregadas sobre os construtores desse tipo. Além disto, modificar o comportamento desta função para valores de um componente específico do tipo `Company` também é bastante simples, bastando modificar a definição sobrecarregada correspondente, sem necessidade de modificar as demais definições. Por exemplo, suponha que queremos definir que dois valores do tipo `Person` são iguais somente se seus nomes são iguais (a definição original considera que um valor do tipo `Person` é igual a outro valor deste tipo se seus

nomes e endereços são iguais). Para isto, bastaria substituir a definição anterior, para valores do tipo `Person`, pela seguinte:

```
overload (==) (P n1 _) (P n2 _) = (n1 == n2);
```

As definições de funções nos exemplos apresentados na Figura 5.4 consistem, como comentamos, de puro código *boilerplate*. Além disso, as funções definidas são específicas para o tipo de dados `Company`, não sendo, portanto, aplicáveis a valores de outros tipos.

Uma possível solução para este problema seria utilizar um tipo de representação estrutural de tipos e definir funções genéricas sobre esse tipo de representação estrutural. Tais funções seriam instanciadas para cada tipo específico, por meio de um isomorfismo entre esse tipo e sua representação estrutural. Esta é, essencialmente, a idéia utilizada em *Generic for the Masses*[Hin04] e *Generic Haskell*, e sofre dos inconvenientes apresentados na Seção 4.4.

Uma solução alternativa é a abordagem de *Scrap your boilerplate*, apresentada na Seção 4.2, que provê uma biblioteca de funções de caminhamento genéricas, que podem ser usadas para definir funções sobre tipos complexos, evitando código *boilerplate*. É fácil perceber que a implementação dessa abordagem requer três recursos. O primeiro é polimorfismo de ordem superior (ou de rank arbitrário), já que funções de caminhamento genérico devem ser parametrizadas por funções polimórficas. O segundo é um mecanismo para definição de funções polimórficas que tenham comportamento específico para determinados tipos. Em SYB, tais funções são definidas utilizando os operadores `mkT` e `mkQ`, que são, por sua vez definidos em termos de um mecanismo de coerção de tipos potencialmente inseguro. Esse recurso de extensão de funções, utilizado pelo SYB, pode também ser visto como um mecanismo de especialização de funções polimórficas, como na visão de Generic Haskell, e será chamado, doravante de definição de *funções polimórficas especializadas*.

Finalmente, um terceiro recurso desejável é um mecanismo de derivação automática dessas funções de caminhamento genérico, para que não tenham que ser definidas pelo programador para cada novo tipo, por meio de definições sobrecarregadas.

Na Seção 5.1.1, vimos que definições sobrecarregadas sobrepostas provêm funcionalidade semelhante a esse mecanismo de definições de funções polimórficas especializadas. Portanto, é interessante investigar como esse mecanismo pode ser usado para esse fim, evitando a possibilidade de coersões de tipo que possam ocasionar erro de tipo em tempo de execução. Note entretanto que, considerando a possibilidade de uso de funções polimórficas especializadas como argumentos de funções, a definição particular a ser usada em uma aplicação desta função não pode ser determinada estáticamente, mas apenas dinâmicamente. Portanto, embora a idéia de definição de instâncias sobre-

postas possa ser utilizada, a semântica (ou implementação) dessas definições deve ser redefinda.

Antes de reapresentar os exemplos anteriores codificados em uma linguagem baseada no sistema CT estendido com os recursos mencionados anteriormente, apresentamos, na seção a seguir, outros exemplos que ilustram a utilidade de polimorfismo de *rank* arbitrário. Em seguida, discutimos como as extensões propostas devem afetar o sistema CT atual.

5.2.1 Polimorfismo de *rank* arbitrário

Considere o seguinte programa:

```
reverse xs = rev xs [] where rev [] ys = ys
                             rev (x:xs) ys = rev xs (x:ys)
foo = let
        f x = (x [True,False], x ['a', 'b'])
      in f reverse
```

Na definição de `f`, a função `x`, é aplicada a uma lista de valores booleanos e a uma lista de caracteres — aparentemente, este código não possui erros de tipos, uma vez que o tipo inferido para `reverse`, neste contexto, é $\forall a.[a] \rightarrow [a]$, isto é, `reverse` deve se comportar de maneira idêntica sobre listas de qualquer tipo.

Todavia, a definição de `foo` é rejeitada pelo sistema CT, uma vez que este estende o sistema de tipos de Damas-Milner, o qual impõe a restrição de que *argumentos de função (como* `x`*) não podem ter tipo polimórfico* (embora uma função polimórfica possam ser parâmetro de uma determinada função, desde que usada no corpo dessa função com tipo monomórfico).

Em um sistema de tipos que trate polimorfismo de *rank* superior, o tipo de `f` é:

```
f ::  (∀ a.[a]→[a])→ ([Bool], [Char])
```

Observe que, neste tipo, a quantificação universal (e a variável quantificada, a) está restrita ao (primeiro) argumento.

Embora possa parecer que polimorfismo de *rank* superior não constitua uma extensão muito útil ao sistema de tipos de Damas-Milner, funções polimórficas de *rank* superior aparecem com freqüência na literatura. Por exemplo:

- *Encapsulamento*: A mônada de estado [LJ95] utiliza a função `runST` de tipo:

  ```
  runST ::  ∀ a.  (∀ s.ST s a)→ a
  ```

A idéia da função `runST` é garantir que uma computação que realiza a atualização de estado, de tipo `ST s a`, seja encapsulada, retornando um valor de tipo `a`. Observe que a variável de tipo `s`, não é visível fora do primeiro parâmetro de `runST`, permitindo que o tipo desta computação seja encapsulado no argumento de `runST`.

- *Programação genérica*: O polimorfismo de *rank* arbitrário é necessário em programação genérica. Por exempo, a abordagem SYB faz uso intensivo de funções com tipos de *rank* superior, como:

  ```
  gmapT :: ∀ a. Data a ⇒ (∀ b. Data b ⇒ b → b) → a → a
  ```

  Abordagens baseadas no trabalho de Hinze[Hin00a, Hin00b] - como Generic Haskell - fazem uso de tipos de *rank* superior para expressar o resultado de seu processo de especialização.

- *Invariantes*: Diversos autores têm explorado a idéia de usar o sistema de tipos para codificar invariantes de tipos de dados, utilizando tipos de dados não regulares[BP99, Oka98]. Por exemplo, em [BP99] é utilizado o seguinte tipo algébrico para representar termos do λ-cálculo:

  ```
  data Term v = Var v |App(Term v) (Term v) |(Lam (Term (Incr v)))
  Incr v = Zero | Succ v
  ```

  O tipo de `fold` para este tipo seria:

  ```
  foldT :: (∀ a. a→ n a)
        → (∀ a. n a→ n a→ n a)
        → (∀ a. n (Incr a)→ n a)
        → Term b → n b
  ```

  Para tipos de dados não regulares, funções como `map` e `fold` só podem ser definidas se o sistema de tipos provê suporte a tipos de *rank* superior.

O compilador GHC provê suporte a tipos de *rank* arbitrário, desde que o programador anote explicitamente o tipo de funções com parâmetros polimórficos. Tais anotações de tipo são necessárias, uma vez que a inferência de tipos de *rank* superior é indecidível [KW94, Wel94].

Em Haskell, tipos de *rank* superior podem incluir restrições de tipo, especificada na forma de restrições de classe. Considere, novamente, o tipo de `gmapT`:

```
gmapT :: ∀ a. Data a ⇒ (∀ b. Data b ⇒ b → b) → a → a
```

Esta função exige que o tipo do argumento para a função correspondente ao primeiro parâmetro de `gmapT` seja uma instância da classe `Data`.

5.2.2 Polimorfismo de ordem superior no sistema CT

Para expressar tipos como o de `gmapT` no sistema CT, além de estender este sistema de tipos com suporte a polimorfismo de *rank* arbitrário, devem ser permitidas restrições de tipo com tipo polimórfico de *rank* arbitrário. Considere, como exemplo, o tipo e a definição da função `everywhere`, que aplica uma função polimórfica, fornecida como parâmetro, a todos os componentes de uma estrutura de dados:

```
type GenericT = ∀ a. Data a ⇒ a → a
everywhere :: GenericT → GenericT
everywhere f = f . gmapT (everywhere f)
```

A definição desta função, usando o sistema CT estendido, seria similar à anterior, exceto pelas restrições no tipo desta função, que seriam conforme mostrado a seguir:

```
type GenericT = ∀a.{gmapT::(∀b.b→b) → a → a} ⇒ a → a
everywhere :: GenericT → GenericT
everywhere f = f . gmapT (everywhere f)
```

A restrição {`gmapT::(∀b.b→b) → a → a`} indica que um símbolo que possua o tipo `GenericT` pode ser usado em qualquer contexto onde exista uma definição de `gmapT` com este tipo.

Essas extensões necessárias para que o sistema CT suporte uma definição de uma função como `everywhere` ainda não foram realizadas, sendo objeto de trabalho futuro.

5.2.3 Funções polimórficas especializadas no sistema CT

Considere a definição da função `incS`, utilizando instâncias sobrepostas, apresentada anteriormente:

```
overload incS v (S s) = S (s * (1 + v))

overload incS _ x = x
```

O tipo inferido pelo sistema CT para `incS`, no contexto das definições acima, seria:

```
incS :: ∀ b. {incS :: Float → b → b}. Float → b → b
```

Seria entretanto necessário modificar o sistema CT, de maneira que o tipo inferido para `incS`, a partir das definições acima, fosse apenas: `incS :: ∀ b. Float → b → b`, indicando que `incS v`, onde `v` tem tipo `Float`, pode ser aplicada a valores de qualquer tipo, sem restrição.

Além disso, é necessário modificar a semântica (ou imlementação) dessas definições, para que funções polimórficas especializadas possam poder ser argumentos de funções. No caso em que funções definidas por meio de instâncicas sobrepostas não podem ser passadas como parâmetro, mas apenas usadas na definição de outras funções, a resolução de sobrecarga é feita estaticamente, e tais funções podem ser implementadas como funções que têm um dicionário como parâmetro adicional, tal como na implementação desenvolvida neste trabalho. No caso em que tais funções podem ser passadas como parâmetro, a resolução da sobrecarga só poderá ser feita em tempo de execução, possivelmente implementando essas funções como funções parametrizadas sobre tipos (polimorfismo de ordem superior, ou sistema F).

5.2.4 Exemplos utilizando as extensões propostas

Nos exemplos a seguir, adotamos a seguinte forma para uma *função polimórfica especializada* `f`, em uma linguagem baseada no sistema CT estendido para maior simplicidade supomos que cada definição sobreposta de `f` consiste em uma única equação:

```
f :: σ where
f = e1
 ⋮
f = en
```

Nesta definição, como dissemos anteriormente, o tipo σ deve ser polimórfico pelo menos uma das definições de `f` deve ter exatamente o tipo σ e o tipo de cada uma das demais definições deve ser uma instância do tipo σ.

Por exemplo, a definição de `incS` seria:

```
incS :: ∀ b. Float→ b→ b where
incS v (S s) = S (v * (1.0 + s))
incS _ x = x
```

A função `increaseSalary` seria definida como na Figura 5.5, utilizando a função `everywhere`, de maneira análoga a sua definição em SYB (veja Figura 4.7).

A função `salaryBill` é também definida de modo análogo a sua definição em SYB. Assim como `increaseSalary`, esta definição utiliza funções com tipo de *rank* superior, ou seja, parametrizadas por uma função polimórfica especializada.

```
type GenericT = ∀a.  {gmapT::(∀ b.  b→ b)→ a→ a}.  a→ a

everywhere ::  GenericT → GenericT
everywhere f = f .  gmapT (everywhere f)

increaseSalary ::  Float → Company → Company
increaseSalary v = everywhere (incS v)

incS ::  ∀ b.  Float→ b→ b where
incS v (S s) = S (v * (1.0 + s))
incS _ x = x
```

Figura 5.5: Definindo increaseSalary usando as extensões propostas ao Sistema CT

```
type GenericQ c = ∀a.  {gmapQ ::  (∀ b.  b → c) → a → [c]} ⇒ a → c

everything ::  (c → c → c) → GenericQ c → GenericQ c
everything k f x = foldl k (f x) (gmapQ (everything k f) x)

salaryBill ::  Company -> Float
salaryBill = everything (+) salary

salary ::  ∀a.  a → Float where
salary (S s) = s
salary _ = 0
```

O último exemplo define a igualdade genérica, também de maneira similar a sua definição em SYB (veja Seção 4.2.1). Ao contrário dos exemplos anteriores, neste caso não é utilizada nenhuma função polimórfica especializada.

```
type GenericQ1 c = ∀a.  {gmapQ ::  (∀ b.  b → c) → a → [c],
                         gApplyQ ::  [GQ c] → a → [c]} .  a → c
data GQ c = GQ (GenericQ1 c)
gzipWithQ ::  GenericQ1 (GenericQ1 c) → GenericQ1 (GenericQ1 [c])
gzipWithQ f t1 t2 = gApplyQ(gmapQ (λ x → GQ (f x)) t1)t2)

geq1 ::  ∀ a b.{gzipWithQ ::  GenericQ1 (GenericQ1 c)
                           → GenericQ1 (GenericQ1 [c]),
                toConstr::a→ Constr,
                toConstr::b→ Constr}.
        a → b → Bool
```

```
geq1 x y = (toConstr x) == (toConstr y) &&
                        and(gzipWithQ geq1 x y)

geq :: ∀ a.{geq1 :: ∀ a. a→ a→ Bool}.a→ a→ Bool
geq = geq1
```

5.2.5 Conclusões

Nesta seção, discutimos as extensões requeridas para suporte a programação genérica no sistema CT, segundo uma abordagem semelhante a SYB.

A principal vantagem da abordagem proposta em relação ao SYB é a ausência de operadores de coerção de tipos. A utilização de funções polimórficas especializadas permitirá obter a mesma funcionalidade provida por esses operadores, sendo a definição de tais funções mais intuitiva e sua utilização não sujeita a erros de tipo em tempo de execução.

Outra vantagem da abordagem proposta é que não haveria necessidade de codificar classes de tipos similares a `Typeable` para permitir a representação de tipos usados na implementação da coerção de tipos. Por exemplo, para representar o tipo `(a,b)` em SYB, devemos utilizar a seguinte classe:

```
class Typeable2 t where
      typeOf2 :: t a b → TypeRep
```

Em SYB, para cada aridade de construtor de tipo há necessidade de uma classe e das respectivas instâncias.

Em [LP05], é apresentada uma estratégia que permite a definição de funções genéricas abertas[2]. Esta abordagem utiliza uma técnica para codificar variáveis que representam classes de tipo. Com isso, a extensibilidade é obtida por meio do próprio mecanismo de classes de tipo e instâncias de Haskell. O estudo de mecanismo mais adequado para suporte à definição de funções abertas, constitui objeto de trabalho futuro.

[2]Uma função é dita ser aberta se permite que novos casos sejam adicionados em diferentes partes do programa[LH06].

Capítulo 6

Implementação

Este capítulo apresenta aspectos da integração *front-end Haskell-CT*, implementado em [Vas04], com o *back-end* do compilador Haskell GHC, resultando em um protótipo para uma linguagem simples – *Haskell-CT*, baseada em Haskell, mas utilizando o sistema *CT*. O *front-end* realiza a análise sintática e a inferência / verificação de tipos para um subconjunto da linguagem *Haskell*. O subconjunto considerado não utiliza declarações de classes e instâncias, uma vez que é baseado no sistema de tipos *CT*, no qual declarações de classes não são requeridas para definição de símbolos sobrecarregados. Além disso, algumas construções sintáticas, presentes em Haskell, também não foram incluídas. Por exemplo, padrões `(n + k)` para tipos numéricos, definição de listas usando *list comprehension* e regras de leiaute.

Para a construção do protótipo, optou-se pela utilização do *back-end* do compilador *GHC*. O processo de integração consistiu na conversão do tipo de dados gerado pelo *front-end* do *CT* para o tipo de dados utilizado internamente pelo GHC, para representação da saída do *front-end* deste compilador.

O restante deste capítulo está organizado da seguinte maneira. Primeiramente, é apresentada uma breve descrição da estrutura do compilador GHC. Em seguida, são apresentadas descrições dos tipos de dados gerados pelo *front-end* do compilador e do tipo de dados gerado pelo *type-checker* do GHC. Finalmente, são descritos o processo de conversão entre os tipos de dados, os módulos utilizados para esta tarefa e as adaptações realizadas no código fonte do compilador GHC.

6.1 O compilador GHC

O compilador GHC (Glasgow Haskell Compiler) representa o estado da arte em termos de compilador e interpretador para a linguagem Haskell. O GHC é um compilador de código fonte livre, desenvolvido e mantido por um grupo de pesquisadores.

O objetivo desta seção é apresentar a arquitetura de implementação do GHC, com maior ênfase em seu *front-end* e nos tipos algébricos gerados nesta etapa. A partir da compreensão da arquitetura deste compilador, foi possível realizar a integração de seu *back-end* ao *front-end* Haskell-CT.

6.1.1 Pipeline do compilação

O processo de compilação do GHC é organizado como um *pipeline*. Durante a compilação de um módulo, o GHC chama outros programas e gera uma série de arquivos intermediários, que servem de entrada para o próximo estágio deste *pipeline*. Todo este processo é organizado pelo arquivo fonte:

`compiler/main/DriverPipeline.lhs`

Para ilustrar o funcionamento deste processo de compilação, suponha que o módulo *Teste.hs*, ou *Teste.lhs* (no caso de *literate script*) seja fornecido para a compilação pelo GHC. Os seguintes passos são realizados:

1. Execuçao do pré-processador **unlit**: Este realiza a remoção de marcações de literate script. Após a execução deste pré-processador, é gerado o arquivo *Teste.lpp*. Este pré-processador encontra-se na pasta *utils* na estrutura de diretórios do compilador GHC.

2. Execução do pré-processador **C**: Este passo é executado apenas se a opção *-cpp* for fornecida na linha de comando. Após a execução deste passo é gerado o arquivo *Teste.hspp*.

3. Execução do compilador propriamente dito: Este passo, ao contrário dos anteriores, não inicia um processo a parte. Ele consiste apenas de uma chamada de função Haskell que gerará um arquivo de interface *Teste.hi*, e dependendo dos flags fornecidos, serão gerados os seguintes arquivos:

   - Código Assembly: utilizando o flag -S é gerado o arquivo *Teste.s* que é composto de código de montagem.
   - Código C: utilizando o flag -fvia-C é gerado o arquivo *Teste.hc* que é composto de código C.
   - Código C –: utilizando o flag -fcmm é gerado o arquivo *Teste.cmm* que é composto de código C–.

4. No caso do uso do flag -fvia-C:

   - Execução do compilador `C` gerando o arquivo *Teste.raw_s*

- Execução do `Evil Mangler`, gerador de código de montagem, que gera o arquivo *Teste.s*

5. Se o flag -split-objs for fornecido, será executado o **splitter** sobre o arquivo *Teste.s*. O **splitter** dividirá este arquivo em diversos arquivos menores para que o ligador estático não realize a ligação de código morto.

6. Execução do **assembler** sobre *Teste.s*, ou sobre cada arquivo gerado pelo **splitter**.

6.1.2 Fases do compilador

Independente de qualquer ferramenta externa utilizada, a organização interna do compilador GHC é organizada em fases, sendo o programa de entrada submetido a uma série de transformações/otimizações, até a geração do código final. O arquivo `compiler/main/HscMain.hs` gerencia o processo de compilação de um módulo. Em cada passo, o compilador transforma gradualmente tipos algébricos gerados a partir do código fonte em outros tipos resultantes da função realizada em cada passo. O arquivo `HscMain` realiza os seguintes passos para a compilação de um módulo de código fonte Haskell:

- Inicialmente, o programa é convertido para o tipo algébrico `HsModule`. Este tipo representa a estrutura sintática de um módulo Haskell. O tipo `HsModule` é parametrizado pelo tipo que representa um identificador em cada uma das etapas do *front-end* do compilador. Inicialmente, este é parametrizado apenas por um string. Os três primeiros passos (*front-end*) do GHC funcionam da seguinte maneira:

  - O **parser** produz o tipo algébrico `HsModule`, parametrizado pelo tipo `RdrName` que, nessa etapa, é apenas um string, que representa o nome do módulo, não qualificado.
  - O **renamer** transforma o tipo algébrico `HsModule` gerado pelo **parser**, que é parametrizado pelo tipo `RdrName`, em um valor de `HsModule` parametrizado por um valor do tipo `Name`. Inicialmente, o tipo `Name` consiste de um string e um número único que o identifica (representado pelo tipo `Unique`). É tarefa do **renamer** qualificar identificadores que são importados para o módulo atual, isto é, se o identificador `f` é importado de um módulo `M`, todas as ocorrências de `f` são substituídas por `M.f`. Além disso, é tarefa do **renamer** garantir que todas as ocorrências de um determinado identificador compartilhem o mesmo valor do tipo `Unique`, para que o gerador de código possa gerar o código correto para todas essas ocorrências.

  - O **type-checker** transforma o tipo algébrico gerado pelo **renamer** em um `HsModule` parametrizado, agora, pelo tipo `Id`. Um valor do tipo `Id` consiste de um valor do tipo `Name` e uma classificação do identificador ao qual está associado. Além disto, é tarefa do `type-checker` a conversão de classes e instâncias para os tipos `Class` e `Instance` que formam os dicionários para símbolos sobrecarregados. O `type-checker` também realiza a conversão de tipos, construtores de tipos, variáveis de tipos e construtores de dados para valores dos tipos `Type, TyCon, TyVar` e `DataCon` respectivamente. Todos os tipos gerados nesta etapa, são pervasivos por todo o restante do processo de compilação.

Os passos acima descritos são responsáveis por descobrir todo e qualquer erro, de sintaxe ou de tipo, que possa ser encontrado no programa, e relatá-lo ao usuário. Os próximos passos são relativos ao *back-end* do compilador.

- O **deSugarer** (`/compiler/deSugar/Desugar.lhs`) converte o tipo algébrico `HsModule` para um tipo algébrico mais simples, que representa a linguagem intermediária utilizada pelo GHC, a linguagem *core* [Tol]. Esta linguagem é representa pelo tipo `CoreSyn`, que é extremamente simples, consistindo apenas de oito construtores.

- Após este passo, uma série de transformações / otimizações são realizada sobre esta representação intermediária, até o final do processo de compilação. Demais passos do *back-end* não serão apresentados, por estarem fora do escopo deste trabalho.

6.2 Aspectos de Implementação

Nesta seção é descrito o processo de implementação do conversor do tipo gerado pelo *front-end*, que utiliza o sistema CT, para o tipo de dados que é manipulado pelo *back-end* do compilador GHC. Antes de descrever a arquitetura deste conversor, as duas próximas sub-seções apresentam o tipo de dados gerado pelo *front-end* e o tipo esperado pelo *back-end*. Finalmente, é apresentada a arquitetura de implementação do programa que realiza a conversão destes tipos de dados.

6.2.1 Tipo de dados gerado pelo Front-end

Como a implementação do *front-end* deste protótipo é baseada no algoritmo apresentado em [Jon99], são usados os mesmos tipos de dados apresentados no trabalho de Mark Jones, com pequenas adaptações. A representação de um módulo é uma tripla:

```
type Program = (String, [String], BindGroup)
```

onde:

- O primeiro elemento da tripla representa o nome do módulo que está sendo compilado.
- O segundo elemento representa a lista de itens exportados por este módulo.
- O terceiro elemento representa o conjunto de definições de funções, variáveis e tipos algébricos presentes neste módulo.

As definições de funções, variáveis e tipos algébricos presentes em `BindGroup` são divididas em definições explicitamente tipadas e definições implicitamente tipadas. Definições explicitamente tipadas são aquelas que possuem anotações de tipos correspondentes e definições implicitamente tipadas são aquelas em que as anotações de tipos foram omitidas. Um `BindGroup` é representado como um par formado por listas destes dois tipos de definições:

```
type BindGroup = ([Expl], [Impl])
```

Definições presentes em um módulo são formadas por um identificador, que representa o símbolo que está sendo definido, e uma lista de equações que definem este símbolo. Independentemente de ser implicitamente tipada ou explicitamente tipada, toda definição é formada por um identificador e uma lista de equações. A diferença entre as duas é a presença (ou ausência) da anotação de tipo para este símbolo. Sendo assim, a definição da estrutura que representa símbolos explicitamente e implicitamente tipados pode ser feita como a seguir:

```
type Expl = (Id, Type, [Alt])
type Impl = (Id, [Alt])
```

6.2.2 Representando tipos declarados ou inferidos

`Type` representa um tipo, possivelmente qualificado por uma lista de restrições, que limitam os tipos para os quais as variáveis deste tipo podem ser instanciadas. Sendo assim, `Type` consiste de uma lista de restrições associadas a um tipo simples:

```
data Type = Forall (Constrained SimpleType)
```

O tipo `Constrained` representa uma lista de restrições associadas a um tipo. Sua definição é:

```
data Constrained t = [Pred] :=> t
```

Onde `[Pred]` representa a lista de restrições para um tipo `t`. Uma restrição é composta por um identificador, um tipo simples e um valor booleano que indica se esta restrição pode ou não ser removida, ou seja:

```
data Pred = Constraint (Id, SimpleType, Bool)
```

Um tipo simples é representado pelo tipo algébrico `SimpleType`. Este tipo representa variáveis de tipo, aplicações e construtores de tipos. Sua definição é mostrada abaixo:

```
data SimpleType = TVar Tyvar
                | TCon Tycon
                | TAp SimpleType SimpleType
                | TGen Int
```

A definição de cada um dos construtores de dados de `SimpleType` é bem intuitiva; exceto por `TGen`. Este construtor representa variáveis de tipos quantificadas [Jon99].

Os tipos `Tyvar` e `Tycon` são utilizados para representar variáveis e construtores de tipos, respectivamente. Cada variável e construtor de tipo possui o seu *kind* associado, conforme apresentado em [Jon99]. A definição destes é apresentada a seguir:

```
data Tyvar = Tyvar String IneqIndexes Kind
data Tycon = Tycon String Kind
data Kind = Star | KFun Kind Kind
```

6.2.3 Representando equações de uma definição de função

O tipo `Alt` representa uma equação que compõe uma definição de função. Como uma equação é composta por uma série de padrões, no lado esquerdo, e uma expressão, do lado direito, a definição deste tipo torna-se evidente:

```
type Alt = ([Pat], Expr)
```

O tipo `Pat` representa uma definição de padrão utilizada em funções, λ-abstrações e expressões case. Um padrão pode ser uma variável, ou um literal, ou um construtor de dados, ou um caractere coringa (*wildcard*), ou um *as-pattern*, ou um padrão irrefutável (*lazy-pattern*). Sendo assim, a definição do tipo `Pat`, segue diretamente:

```
data Pat = PVar Id
         | PLit Literal
         | PWildcard
         | PAs Id Pat
         | PLazy Pat
```

O tipo `Expr` representa uma expressão que pode ser encontrada no código fonte. Conforme citado anteriormente, nem todo tipo de abreviação sintática definida na sintaxe de Haskell é tratada por este protótipo. Os tipos de expressões válidas, consideradas neste protótipo, são: variáveis, literais, construtores de dados, aplicação, expressões-let, comandos condicionais (*if*), λ-abstrações e expressões case. Disto segue a definição do tipo `Expr`:

```
data Expr = Var Id
          | Lit Literal
          | Const Assump
          | Ap Expr Expr
          | Let BindGroup Expr
          | Lam Alt
          | If Expr Expr Expr
          | Case Expr [(Pat, Expr)]
```

6.2.4 Conversão do tipo de dados gerado pelo Front-end

Apesar de o tipo de dados gerados pelo *parser* do protótipo ser conveniente para a execução do algoritmo de inferência de tipos, este não é adequado para o processo de conversão para o tipo de dados esperado pelo *back-end* do GHC.

Isto se deve ao fato de que a principal tarefa do conversor é, a partir de símbolos sobrecarregados definidos no módulo a ser compilado, gerar os tipos de dados equivalentes a declarações de classes e instâncias. Portanto, é adequado que o tipo algébrico fornecido ao processo de conversão facilite esta tarefa. Para isto foi definido um novo tipo de dados para representar o resultado do *front-end* deste protótipo, cuja definição é mostrada no seguinte trecho de código:

```
data CTFrontEndResult = CTModule String [String] [CTBind]
data CTBind = Overloaded Lcg [Expl]
            | DataType Impl
            | Simple Expl
```

Na definiçao do construtor de dados `CTModule`, os dois primeiros parâmetros possuem a função de representar o nome do módulo compilado e sua respectiva lista de exportações. O terceiro parâmetro (`[CTBind]`) representa o conjunto de definições presentes neste módulo. O tipo `CTBind` representa as possíveis declarações *top-level* presentes no módulo. Este tipo possui três construtores de dados:

- `Overloaded Lcg [Expl]`: Este construtor representa todas as definições de um símbolo que está sobrecarregado no módulo e possui dois parâmetros:

- `Lcg` Representa a *least common generalization* dos tipos de todas as definições do símbolo sobrecarregado. O tipo `Lcg` é apenas um sinônimo para o tipo `Assump`, utilizado pelo *front-end* para representar suposições de tipos para um determinado símbolo.
- `DataType Impl`: Este construtor representa definições *top-level* que não correspondem a declarações de funções, como, por exemplo, declarações de tipos algébricos. O único parâmetro deste construtor é um valor de tipo `Impl`, que representa esta declaração neste módulo.
- `Simple Expl`: Este construtor representa definições de símbolos não sobrecarregados presentes neste módulo. Seu único parâmetro é um elemento de tipo `Expl`, que representa a definição deste símbolo.

6.2.5 O tipo de dados gerado pelo type-checker do GHC

Nesta seção é apresentado o tipo de dados criado pelo GHC após seu processo de inferência de tipos. O tipo de dados gerado pelo *front-end* é `TblGblEnv`. Este tipo consiste em um registro, que representa todas as declarações *top-level* do módulo que está sendo compilado. A seguir é mostrada a definição deste tipo, que consiste em um registro que está contido no módulo `/compiler/typecheck/TcRnTypes.lhs` (presente no código fonte do compilador GHC).

```
data TcGblEnv = TcGblEnv {
                    tcg_mod :: Module,
                    tcg_src :: HscSource,
                    tcg_rdr_env :: GlobalRdrEnv,
                    tcg_fix_env :: FixityEnv,
                    tcg_type_env :: TypeEnv,
                    tcg_type_env_var :: TcRef TypeEnv,
                    tcg_inst_env :: InstEnv,
                    tcg_exports :: NameSet,
                    tcg_imports :: ImportAvails,
                    tcg_dus :: DefUses,
                    tcg_keep :: TcRef NameSet,
                    tcg_inst_uses :: TcRef NameSet,
                    tcg_th_used :: TcRef Bool,
                    tcg_dfun_n :: TcRef Int,
                    tcg_rn_imports :: Maybe[LImportDecl Name],
                    tcg_rn_exports :: Maybe[Located (IE Name)],
```

```
    tcg_rn_decls :: Maybe (HsGroup Name),
    tcg_binds :: LHsBinds Id,
    tcg_deprecs :: Deprecations,
    tcg_insts :: [Instance],
    tcg_rules :: [LRuleDecl Id],
    tcg_fords :: [LForeignDecl Id]
}
```

Uma breve explicação de cada um dos componentes deste tipo de dados é apresentada a seguir:

- `tcg_module :: Module`: Representa o nome do módulo que está sendo compilado.
- `tcg_src :: HscSource`: Representa o tipo de arquivo que está sendo compilado. O GHC pode receber como entrada tanto arquivos de código Haskell ou arquivos que descrevem programas em sua linguagem intermediária.
- `tcg_rdr_env :: GlobalRdrEnv`: Representa o ambiente de declarações *top-level* para o módulo que está sendo compilado.
- `tcg_default :: Maybe [Type]`: Representa tipos utilizados em uma declaração *default* no módulo que está sendo compilado.
- `tcg_fix_env :: FixityEnv`: Representa declarações de precedência de símbolos presentes neste módulo.
- `tcg_type_env :: TypeEnv`: Representa o ambiente global de tipos para este módulo. Todos os construtores de tipos e definições de classes estão presentes neste ambiente.
- `tcg_type_env_var :: TcRef TypeEnv`: É utilizado para armazenar informações necessárias para a inicialização do arquivo de interface para este módulo.
- `tcg_inst_env :: InstEnv`: Ambiente de instâncias definidas neste módulo.
- `tcg_exports :: NameSet`: Representa os símbolos exportados por este módulo.
- `tcg_imports:: ImportAvails`: Armazena informações sobre o que está sendo importado para este módulo.
- `tcg_dus :: DefUses`: Armazena informações sobre o que é definido e o que é utilizado neste módulo.

- `tcg_keep :: TcRef NameSet`: Ambiente que armazena nomes que não devem ser eliminados como código morto durante o processo de otimização de código.
- `tcg_th_used :: TcRef Bool`: Armazena o valor `True`, se a extensão *Template Haskell* estiver sendo utilizada.
- `tcg_rn_imports :: Maybe [LImportDecl Name]`: Importações presentes no módulo.
- `tcg_rn_exports :: Maybe [Located (IE Name)]`: Exportações declaradas neste módulo.
- `tcg_rn_decls :: Maybe (HsGroup Name)`: Declarações que foram qualificadas, durante o processo de *renaming* do GHC.
- `tcg_binds :: LHsBinds Id`: Representação de *bindings* definidos no módulo.
- `tcg_deprecs :: Deprecations`: Representação do que é definido como *depreciado* neste módulo.
- `tcg_insts :: [Instance]`: Declarações de instâncias deste módulo.
- `tcg_rules :: [LRuleDecl Id]`: Regras definidas neste módulo.

6.2.6 Arquitetura da implementação

A implementação do processo de conversão entre as estruturas de dados apresentadas nas seções 6.2.1 e 6.2.5 é composta por seis módulos, além de todo o código do GHC e do *front-end* implementado por [Vas04]:

- `CTFrontEnd.hs`: Este módulo contém a função `ctFrontEnd :: CTFrontEndResult → TcGblEnv`, que inicia o processo de conversão entre as estruturas de dados produzidas pelo *front-end* e o tipo esperado pelo *back-end* do GHC.
- `CTFrontEndData.hs`: Este módulo contém definições de tipos de dados que são utilizados durante o processo de conversão. Neste módulo estão definidos os tipos `CTFrontEndResult` e `CTBind`, apresentados na Seção 6.2.1.
- `TopLevelConverter.hs`: Este módulo é responsável pela conversão de definições *top-level* presentes no módulo que está sendo compilado. Possui funções responsáveis pela conversão de tipos de dados que representam declarações de tipos algébricos e funções não sobrecarregadas. Funções sobrecarregadas são convertidas para classes e instâncias, pelo processo presente no módulo `ClassInstGen.hs`.

- `ClassInstGen.hs`: Este módulo é responsável pela conversão de símbolos sobrecarregados definidos no módulo que está sendo compilado. Detalhes de como esta conversão é realizada são apresentados na Seção 6.2.7.

- `TypeConverter.hs`: Este módulo é responsável pela conversão de tipos.

- `ExprConverter.hs`: Este módulo é responsável pela conversão de expressões que estão presentes nas definições do módulo que está sendo compilado.

- `PatConverter.hs`: Este módulo é responsável pela conversão de padrões presentes em definições de funções, λ-abstrações e expressões case.

Além destes módulos que realizam a conversão entre estes tipos, foram implementados módulos para realizar o *pretty-print* destes tipos, para fins de depuração e testes.

O processo de conversão de `CTFrontEndResult` em `TcGblEnv` é direto, exceto pela conversão de símbolos sobrecarregados. Detalhes de como é realizada a conversão nestes casos são apresentados a seguir.

6.2.7 Tradução de símbolos sobrecarregados

Em Haskell, símbolos sobrecarregados são definidos por meio de declarações de classes e instâncias, conforme apresentado na Seção 3.1. Conforme a semântica de funções sobrecarregadas definida em [WB89], funções sobrecarregadas são traduzidas como funções que recebem um argumento extra, denominado dicionário. Um dicionário de um símbolo sobrecarregado armazena as diversas definições para este símbolo. Nesta seção, mostraremos o processo de geração de dicionários para símbolos sobrecarregados na liguagem *Haskell-CT*. Antes de apresentar esse processo de geração de dicionários, descrevemos, na próxima seção, os tipos de dados envolvidos neste processo.

6.2.7.1 Tipos de representação de dicionários

No compilador GHC, os dicionários corrrespondnetes a símbolos e literais sobrecarregados são representados pelos tipos `Class` e `Instance`, mostrados no trecho de código a seguir:

```
data Class
  = Class {
     classKey  :: Unique,
     className :: Name,
     classTyVars  :: [TyVar],
     classFunDeps :: [FunDep TyVar],
```

```
    classSCTheta :: [PredType],
    classSCSels  :: [Id],
        classInsts :: [Instance]
  }

data Instance
  = Instance {
        is_cls  :: Class,
        is_tys  :: [Type],
        is_dfun :: LHsBinds Id,
        is_ctx :: [PredType],
        is_flag :: OverlapFlag
    }
```

Os componentes do tipo `Class` são explicados a seguir:

- `classKey :: Unique`: É um número, gerado internamente pelo GHC, que identifica a classe de maneira unívoca, durante todo o processo de compilação.
- `className :: Name`: Nome da classe.
- `classTyVars :: [TyVar]`: Variáveis de tipo da classe.
- `classFunDeps :: [FunDep TyVar]`: Dependências funcionais.
- `classSCTheta :: [PredType]`: Lista de super classes.
- `classSCSels :: [Id]`: Lista de identificadores dos símbolos definidos nesta classe.
- `classInsts :: [Instance]`: Lista de instâncias desta classe.

Os componentes do tipo `Instance` são:

- `is_cls :: Class`: Classe à qual pertence esta instância.
- `is_tys :: [Type]`: Lista dos parâmetros dos valores dos parâmetros de tipo desta instância. Como exemplo, na definição de uma instância da classe `Eq` para o tipo `Int`, este campo teria o valor `[Int]`
- `is_dfun :: LHsBinds Id`: Conjunto de equações para as definições de símbolos sobrecarregados para esta instância.
- `is_ctx :: [PredType]`: Restrições presentes na definição desta instância.

- `is_flag :: OverlapFlag`: *Flag* utilizado para indicar se esta instância é sobreposta ou não.

Cada classe possui sua respectiva lista de instâncias e, além disso, possui uma lista de super classes, onde cada super classe é representada por um valor de tipo `PredType`. O tipo `PredType` é definido como:

```
data PredType
        = ClassP Class [Type]
        | IParam (IPName Name) Type
```

O primeiro construtor de `PredType` é utilizado para representar contextos de classes em declarações de classes, instâncias e tipos. Este construtor tem como argumentos uma classe e a lista dos parâmetros de tipo desta classe. Como exemplo, considere a seguinte anotação de tipo:

```
f :: (Eq a) ⇒ a → Int
```

Nesta anotação de tipo, a restrição `Eq a` é representada por um valor do tipo `PredType`, que possuirá uma lista de tipos contendo apenas a variável `a`.

Assim como o tipo `Class`, o tipo `Instance` também possui uma lista de valores do tipo `PredType`, que representa o contexto que pode estar presente na definição de uma instância.

Na representação de dicionários utilizada pelo GHC uma determinada classe ou instância é por um mesmo valor, em todo o processo de compilação. Sendo assim, a estrutura de tipos e dicionários forma um *grafo*, como é ilustrado pelo exemplo a seguir. Considere as seguintes classes e instâncias:

```
class (B a b) ⇒ C a b c where
    c :: a → b → c

class B a b where
    b :: a → b

instance B Int Bool where
    b x = True

instance C Int Bool Bool
    c x y = y
```

A estrutura de dicionários correspondente a estas declarações de classes e instâncias pode ser representada esquematicamente por:

```
dict_class_B = Class {
                    classKey = keyB,
                    className = nameB,
                    classTyVars = [a, b],
                    classFunDeps = [],
                    classSCTheta = [],
                    classSels = [b],
                    classInsts = [b_Int_Bool]
                }
dict_class_B = Instance {
                    is_cls = dict_class_B,
                    is_tys = [Int, Bool],
                    is_dfun = -- equação do símbolo b
                    is_ctx = [],
                    is_flag = NoOverlap
                }
dict_class_C = Class {
                    classKey = keyC,
                    classTyVars = [a, b, c],
                    classFunDeps = [],
                    classSCTheta = [dict_class_B],
                    classSels = [c],
                    classInsts = [c_Int_Bool_Bool]
                 }
c_Int_Bool_Bool = Instance {
                    is_cls = dict_class_C,
                    is_tys = [Int, Bool, Bool],
                    is_dfun = -- equação do símbolo c
                    is_ctx = [],
                    is_flag = NoOverlap
                }
```

Como pode ser observado no trecho do código anterior, existe apenas um valor para a representação de uma classe. Veja que a sub-classe `C` e a instância `B Int Bool` compartilham o mesmo valor, de tipo `Class`, que representa a classe `B`. É óbvio que o trecho de código anterior é apenas uma representação simplificada. Valores de

tipo `Name`, `Id`, `PredType` e `Type` foram não representados utilizando seus respectivos construtores, por não serem necessários para a compreensão da estrutura de dados que representa dicionários.

A próxima seção apresenta detalhes do processo de tradução do tipo de dados que representa definições sobrecarregadas, gerado pelo *front-end* CT, para a representação de dicionários do GHC.

6.2.7.2 Geração de dicionários

Instâncias de uma determinada classe fornecem definições para os símbolos cujos tipos foram anotados nesta classe. Cada uma destas definições deve possuir um tipo que é uma instância do tipo anotado na declaração da classe[1]. Podemos também dizer que o tipo anotado para um símbolo, em uma declaração de classe, é mais geral que o tipo deste símbolo em uma declaração de instância desta classe.

Seja $(\pi, \sigma^{i=1...n}, \{\rho\}^{i=1...n})$ um conjunto de n definições sobrecarregadas para um determinado símbolo π, onde $\sigma^{i=1...n}$ representa o tipo de cada uma destas definições e $\{\rho\}^{i=1...n}$ o conjunto de equações que formam cada uma das n definições de π. Seja $\sigma^{'} = \forall\overline{\alpha}.\overline{\kappa}.\tau$ o *lcg* dos tipos das definições de π. O processo de geração de dicionário correspondente a este conjunto de definições sobrecarregadas é realizado pelos seguintes passos:

- A partir do *lcg* $\sigma^{'} = \forall\overline{\alpha}.\overline{\kappa}.\tau$, é gerado um valor do tipo `Class`, que representa a declaração de classe para π. Isso é feito da seguinte maneira:
  - Inicialmente, é gerado o nome desta nova classe, o qual é obtido a partir do identificador sobrecarregado π, tornando sua primeira letra maiúscula.
  - Para cada restrição $(x : \sigma) \in \overline{\kappa}$, é gerada a seguinte especificação de super classe: $(X\ tv(\sigma))$, onde X corresponde a um identificador de classe de tipos e $tv(\sigma)$ representa as variáveis de tipo presentes no tipo σ, especificado na restrição $(x : \sigma)$. Cada restrição de superclasse é formada por um valor do tipo `PredType`.
  - Em seguida, é gerado um método correspondente ao símbolo π, para esta classe. O identificador deste método é o próprio identificador de π, sendo o tipo τ anotado em sua definição.
- Para cada par σ^i e ρ^i (com $1 \leq i \leq n$) é gerada uma declaração de instância, da seguinte maneira:

[1] Se σ_1 é o tipo anotado para um símbolo em uma classe e σ_2 é o tipo deste símbolo em uma instância desta classe, dizemos que o tipo deste símbolo nesta instância é correto se *unify*(σ_1, σ_2) = S para alguma substituição S.

- Gera-se, a partir de ρ^i, o conjunto de equações correspondentes à definição desta instância do símbolo π.
- Seja $\sigma^i = \forall\overline{\alpha^i}\kappa^i\tau^i$ o tipo desta i-ésima instância de π. Para a determinação de como as variáveis de tipo, presentes no valor de tipo `Class` criado para o símbolo π, serão instanciadas nesta declaração, primeiramente é obtida uma substituição S, tal que $S\sigma^{'} = \sigma^i$, onde $\sigma^{'}$ corresponde ao *lcg* dos tipos das definições de π. Em seguida, aplica-se S ao conjunto de variáveis de tipo de $\sigma^{'}$, determinando-se assim, como cada uma destas variáveis de tipo deve ser instanciada nesta definição de π. Os tipos resultantes são convertidos em valores de tipo `Type`, que são utilizados internamente pelo GHC.
- Para cada par de restrições $(x : \tau) \in \kappa^i$, é obtida uma substituição S tal que $S\tau^x = \tau$, onde τ^x corresponde ao tipo simples presente no *lcg* dos tipos de todas as definições do símbolo x. O elemento de contexto correspondente à restrição $(x : \tau)$, adicionado à declaração desta instância, é $(X\ Stv(\tau^x))$, onde X corresponde ao identificador do valor do tipo `Class`, para o símbolo x, e $Stv(\tau^x)$ corresponde à aplicação da substituição S ao conjunto de variáveis de tipo presentes no *lcg* de x.

Para uma melhor compreensão do processo de conversão, são apresentados a seguir dois exemplos de código fonte e suas respectivas conversões, utilizando os passos acima descritos. Cabe ressaltar, novamente, que estes exemplos são meramente ilustrativos, pois o código Haskell correspondente às declarações de classes e instâncias não é gerado. O que é produzido durante a etapa de conversão são as estruturas de dados que representam os dicionários de sobrecarga correspondentes a essas definições.

6.2.7.3 Exemplos de conversão de símbolos sobrecarregados

Nos dois exemplos seguintes, são mostradas as definições de dois símbolos sobrecarregados, `length` e `negate`, e em seguida as respectivas declarações de classes e instâncias. Como primeiro exemplo, considere a definição do símbolo `negate`:

```
overload negate True = False;
         negate False = True;

overload negate n = (-1) * n;
```

Evidentemente, estas definições possuem os seguintes tipos (supondo que `*` possua o tipo `Int` → `Int` → `Int`):

```
negate :: Bool → Bool
negate :: Int → Int
```

Portanto, neste contexto, o símbolo `negate` possui o seguinte *lcg*:

```
negate :: {negate :: a → a → a } . a → a → a
```

Conforme apresentado anteriormente, uma definição de classe para `negate` é gerada a partir do *lcg* dos tipos de todas as definições de `negate`. A definição de classe gerada para o símbolo `negate`, com as definições acima, seria:

```
class Negate a where
    negate :: a → a → a
```

Apesar do *lcg* das definições de `negate` possuir a restrição `negate :: a → a → a`, não é gerada nenhuma especificação de superclasse, uma vez que a restrição presente no *lcg* é referente ao próprio símbolo `negate`, neste caso, esta restrição é ignorada e nenhuma especificação de super-classe é gerada.

Em seguida, para cada definição de `negate`, são geradas as seguintes declarações de instância:

```
instance Negate Bool where
    negate True = False
    negate False = True


instance Negate Int where
    negate n = (-1) * n
```

Para obter o tipo para o qual a variável de tipo `a` será instanciada, para cada uma destas instâncias é determinada uma substituição S que torne o *lcg* igual ao tipo da instância; em seguida, esta substituição é aplicada as variáveis de tipo do *lcg* deste símbolo. Evidentemente, as seguintes substituições tornam o *lcg* de `negate` igual ao tipo de cada uma de suas definições:

Tipo	Substituição
`negate :: Bool → Bool`	$S_1 = \{a \mapsto Bool\}$
`negate :: Int → Int`	$S_2 = \{a \mapsto Int\}$

Aplicando cada uma destas substituições à variável de tipo `a` presente no *lcg* de `negate`, obtém-se como esta deve ser instanciada em cada uma das declarações de instância do símbolo `negate`.

O segundo exemplo trata da conversão das definições sobrecarregadas da função `length`, para os tipos `a`, `[a]`, `Maybe a` e `Tree a`. As definições de `length` para estes tipos são apresentadas abaixo:

```
overload length _ = 1;

overload length [] = 0;
         length (x:xs) = (length x) + length xs;

overload length Nothing = 0;
         length (Just x) = length x;

overload length Leaf = 0;
         length (Branch x l r) = (length x) + (length l) + (length r);
```

Os tipos de cada uma destas definições são:

```
length ::  a → Int
length ::  {length::a b → Int} ⇒ Maybe a b → Int
length ::  {length::a b → Int} ⇒ [a b] → Int
length ::  {length::a b → Int} ⇒ Tree a b → Int
```

e o *lcg* destes tipos é:

```
length ::  {length::a b → Int} .  a b → Int
```

Portanto, segundo o processo apresentado na Seção 6.2.7, a seguinte classe é gerada para estas definições:

```
class Length a b where
   length ::  a b → Int
```

Para a geração da instância correspondente à definição de tipo `a → Int`, o processo é similar ao descrito anteriormente para as instâncias do símbolo `negate`. O código correspondente a esta instância de `length` é mostrado a seguir:

```
instance Length a where
   length _ = 1
```

Para as demais instâncias deste símbolo, devemos levar em consideração a presença da restrição `length::a b → Int`. Esta restrição faz que seja adicionada à declaração

de uma instância de `length` um contexto que as variáveis de tipo presentes na cabeça da instância[2] devem satisfazer.

Conforme apresentado na Seção 6.2.7, a presença de uma restrição $(x : \sigma)$ no tipo de uma definição sobrecarregada de um símbolo `y`, resulta na inclusão de um item de contexto na definição de instância correspondente a esta definição. No trecho de código anterior, as instâncias de `length` para os tipos `Maybe`, `[a]` e `Tree a` possuem a restrição `length::a b` $\rightarrow$ `Int`. Esta restrição é convertida em um item de contexto da seguinte maneira: inicialmente é obtida uma substituição S tal que S(`length::a` $\rightarrow$ `Int`)=`length::a b` $\rightarrow$ `Int`. A substuição S que torna o *lcg* de `length` igual à restrição é:

$S = \{a \mapsto (a\ b)\}$

Usando esta substituição, é gerado o item de contexto `Length (a b)`, para cada uma das instâncias da classe `Length (a b)`. A seguir são mostradas as demais declarações de instâncias para `length`:

```
instance (Length (a b)) ⇒ Length [a b]
    length [] = 0
    length (x:xs) = (length x) + length xs

instance (Length (a b)) ⇒ Length Maybe (a b)
    length Nothing = 0
    length (Just x) = (length x)

instance (Length (a b)) ⇒ Length Tree (a b)
    length Leaf = 0
    length (Branch x l r) = (length x) + (length l) + (length r)
```

6.2.8 Modificações realizadas no código fonte do compilador GHC

A principal modificação realizada no código fonte do compilador GHC foi a completa substituição de seu *front-end* pelo *front-end* que utiliza o sistema CT. Conforme descrito na Seção 6.1.2, todo o pipeline de compilação de um único módulo está contido em `/compiler/main/HscMain.lhs`. Neste módulo foi necessária a alteração de uma

[2]Em Haskell, toda declaração de instância deve possuir a seguinte forma: `instance cx` $\Rightarrow$ `C (T` $u_1 \ldots u_k$`) where {d}`[Jon03]. Denomina-se cabeça de uma declaração de instância o componente C (T $u_1 \ldots u_k$), onde `C` é um identificador de classe e (`T` $u_1 \ldots u_k$) um tipo.

única função, `hscFrontEnd`, que corresponde ao *front-end* utilizado pelo GHC para a compilação de arquivos de código fonte Haskell. Nesta função, substituímos toda a etapa de *parsing* e verificação / inferência de tipos realizada pelo GHC, pelos equivalentes implementados para um compilador que utiliza o sistema CT.

6.3 Limitações

Nesta seção apresentamos as limitações da implementação, descrita neste capítulo, do compilador Haskell-CT.

A primeira limitação, desta versão do compilador, é que este não é capaz de lidar com todos os programas tipáveis pelo sistema CT. Ao contrário de Haskell[3], o sistema CT considera corretos programas que utilizam símbolos que formam uma hierarquia cíclica de instâncias. Como exemplo, considere o trecho de código, apresentado na figura 6.1. Neste trecho de código, temos que as funções primitivas `primMinusInt`, `primEqInt`, `primMinusFloat`, `primEqFloat` possuem os tipos:

```
primMinusInt ::  Int → Int → Int
primEqInt ::  Int → Int→ Bool
primMinusFloat ::  Float → Float → Float
primEqFloat ::  Float → Float→ Bool
```

Além disso, considere também que existam as seguintes definições de `coerce` neste contexto:

```
coerce ::  Int → Float
coerce ::  Int → Int
```

Os tipos inferidos para `odd` e `even`, neste contexto, são apresentados na figura 6.2.

Como pode ser observado nos tipos inferidos para `odd` e `even`, apresentados na figura 6.2, estes possuem restrições mutuamente recursivas. Para estes casos, o processo de geração de dicionários apresentado neste capítulo, não é capaz de produzir resultados que sejam adequados para a utilização do gerador de código do compilador Haskell GHC. Nestas situações, ocorre a interrupção imediata do processo de geração de código para o programa em questão devido a um erro em tempo de execução ocasionado pela hierarquia cíclica de classes gerada.

[3]Em Haskell[Jon03], não é permitido definir classes que formam uma hierarquia cíclica, isto é, a relação de super-classes introduzida por declarações de classes de tipos deve formar um grafo direcionado acíclico.

```
overload (-)  = primMinusInt;
overload (==) = primEqInt;
overload (-)  = primMinusFloat;
overload (==) = primEqFloat;

overload odd n = if n == (coerce 0) then False else even (n - coerce 1);

overload even n = if n == (coerce 0) then True else odd (n - (coerce 1));

overload odd n = False;
overload even n = True;
```

Figura 6.1: Trecho de código com restrições mutuamente recursivas.

```
odd ::  {==::a → a→ Bool, coerce::Int → a, even::a → Bool,
          -::a → a → a}.Int → Bool
odd ::  a → Bool
even ::  {==::a → a → Bool, coerce::Int →a, odd::a → Bool,
            -::a → a → a}.Int → Bool
even ::  a → Bool
```

Figura 6.2: Tipos com restrições mutuamente recursivas.

6.4 Conclusão

Neste capítulo foram apresentadas as decisões de projeto e implementação para a integração do *front-end Haskell-CT* com o *back-end* do compilador Haskell GHC. Inicialmente foi apresentada uma introdução à arquitetura de implementação deste compilador e de seus principais tipos de dados. Em seguida apresentamos, de maneira suscinta, a organização do tipo de dados Haskell gerado pelo *front-end Haskell-CT* e de como estes são convertidos em estruturas de dados internas utilizadas pelo compilador GHC. Apresentamos, também, as modificações realizadas no código deste compilador Haskell para a integração do *back-end* com o novo *front-end*. Finalmente, consideramos as limitações atuais do protótipo de compilador *Haskell-CT*, que impedem que este seja utilizado para implementar funções com restrições mutuamente recursivas.

Capítulo 7

Conclusão

Neste trabalho, identificamos extensões necessárias ao sistema CT, para prover suporte a programação genérica no estilo de Scrap your boilerplate, porém de maneira adequada. Ou seja, sem necessidade de realização de conversões de tipos, que podem ocasionar erros durante a execução de programas. Essas extensões são: *polimorfismo de rank arbitrário* e um mecanismo para definição de *funções polimórficas especializadas*. Em uma linguagem baseada no sistema CT com tais extensões, será possível funções sobre tipos de dados complexos, que operam apenas sobre determinados componentes desse tipo, sem que isso envolva código repetitivo para travessia da estrutura de valores do tipo complexo.

Além dessas duas extensões, é também desejável prover, na linguagem, uma biblioteca de funções genéricas comumente usadas (tais como `everywhere` e `gmaT`), definidas em termos de funções genérias mais fundamentais (por exemplo, `gfold`), e um mecanismo para derivação automática dessas funções genéricas fundamentais para qualquer tipo de dado algébrico definido pelo programador.

Como subsídio para a avaliação das extensões requeridas ao sistema CT para suporte a programação genérica, foram examinadas as abordagens para Programação Genérica em Haskell atualmente existentes, e realizada uma comparação entre as duas abordagens mais relevantes: Scrap your boilerplate e Generic Haskell (Capítulo 4).

Também como subsídio para esta avaliação, foi implementada a integração do *front-end* do sistema CT, descrito em [Vas04], com o *back-end* do compilador Haskell GHC, de maneira a obter o primeiro compilador para uma linguagem baseada no sistema CT. Além disso, foi modificada a política de sobrecarga do Sistema CT, para permitir definições sobrecarregadas sobrepostas de um mesmo símbolo (ou seja, definições sobrecarregadas cujos tipos podem ser unificados).

As principais contribuições deste trabalho são:

- Identificação de extensões necessárias ao sistema CT para suporte a programação

genérica de maneira adequada: sem necessidade de realização de conversões de tipos que podem ocasionar erros durante a execução de programas, sem a necessidade de definir funções baseadas em indução estrutural e sem a necessidade de uso de código *boilerplate* (Capítulo 5).

- Apresentação das principais abordagens existentes para Programação Genérica em Haskell e uma comparação entre as abordagens mais relevantes (Capítulo 4).
- Integração do *front-end* implementado em [Vas04] com o *back-end* do compilador Haskell GHC (Capítulo 6). Como resultado desse trabalho obtivemos o primeiro compilador para uma linguagem baseada no sistema CT.
- Definição e implementação de uma extensão da política de sobrecarga do Sistema CT para permitir definições de símbolos sobrecarregados com tipos sobrepostos (Capítulo 5).

Possíveis trabalhos futuros envolvem:

- Extensão do sistema de tipos e do algoritmo de inferência de tipos para suporte ao polimorfismo de *rank arbitrário* e para definição de funções polimórficas especializadas.
- Especificação da semântica de funções polimórficas especializadas e implementação dessas funções, levando em conta que resolução da sobrecarga, nesse caso, deve ser feita em tempo de execução.
- Estudo de mecanismos para permitir a derivação automática de código *b*oilerplate. Uma possível continuação deste trabalho, seria o desenvolvimento de uma técnica de derivação de código similar ao obtido quando é usada a cláusula *deriving* em Haskell, porém de forma mais abrangente, com menos limitações.
- Estudo de mecanismo para suporte à definição de funções abertas (uma função é dita ser aberta se permite que novas equações sejam adicionadas a sua definição em diferentes pontos do programa). Esse recurso é interessante para permitir um desenvolvimento incremental de programas.

Apêndice A

Definições da função Map

A.1 Introdução

Em Haskell, a função *map*, para o tipo de dados lista, faz parte de sua biblioteca padrão. A expressão `map f xs` representa a lista obtida aplicando a função `f` a cada um dos elementos de `xs`, isto é:

```
map f [x1, x2, ..., xn] == [f x1, f x2, ..., f xn]
map f [x1, x2, ...] == [f x1, f x2, ...]
```

Aplicar uma determinada função a todos os elementos de uma estrutura de dados é uma tarefa comum em desenvolvimento de programas utilizando uma linguagem funcional. A definição da função `map`, para um determinado tipo algébrico é direta e segue a estrutura deste. Considere como exemplos as seguintes definições de `map` para árvores e `Maybe`:

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

mapTree f Leaf = Leaf
mapTree f (Node x l r) = Node (f x) (mapTree f l) (mapTree f r)

data Maybe a = Nothing | Just a

mapMaybe f Nothing = Nothing
mapMaybe f (Just x) = Just (f x)
```

Como pode ser observado, a definição da função `map` para um determinado tipo algébrico consiste, em grande parte, de puro código *boilerplate*. Portanto, técnicas de programação genérica podem se mostrar úteis para o desenvolvimento desta função

para tipos de dados arbitrários. O objetivo deste apêndice é apresentar a definição da função `map` em *Scrap your boilerplate*, Generic Haskell e Sistema CT.

A.2 Definições da função Map

A.2.1 *Scrap your boilerplate*

A abordagem *SYB* permite a definição de caminhamentos sobre tipos de dados complexos aplicando uma determinada função a valores de tipos de dados arbitrários. Em parte este é o comportamento esperado por uma função similar a `map`, que aplica uma função a cada componente de estrutura de dados. Porém, não é possível definir uma função que é parametrizada sobre um construtor de tipos, ou seja, não é possível a definição de uma função de tipo

$$\forall\ a\ b\ f.(a \to b) \to f\ a \to f\ b,$$

onde os argumentos da variável de tipo f são modificados e a função é parametricamente polimórfica sobre a e b. Isto se deve ao fato de que *SYB* utiliza, implicitamente, uma representação estrutural de valores. Como construtores de tipos não são representados por valores do tipo usado para representação estrutural de tipos, não é possível construir uma representação de valores para estes.

A.2.2 Generic Haskell

Antes de apresentar a definição da função `map` em Generic Haskell, é conveniente observarmos novamente a definição da identidade genérica, apresentada na Figura 4.15:

```
id⟨a::⋆⟩ ::  (id) ⇒ a → a
id⟨Int⟩ x = x
id⟨Char⟩ x = x
id⟨Unit⟩ x = x
id⟨Sum a b⟩ (Inl x) = Inl(id ⟨a⟩ x)
id⟨Sum a b⟩ (Inr x) = Inr(id ⟨b⟩ x)
id⟨Prod a b⟩ (x × y) = (id ⟨a⟩ x)×(id ⟨b⟩ y)
```

Observe que as equações de definição de `id` para os tipos estruturais soma e produto geram uma dependência, uma vez que `id` é usada recursivamente nestas equações, aplicada a um tipo diferente daquele que ocorre do lado esquerdo da equação.

Usando o mecasnimo de defnição local de especialização de funções diponível em Generic Haskell (`extends`), é possível definir uma função para incrementar todos os

elementos de uma lista por um determinado valor, especializando a função `id` definida acima:

```
incBy x v = let id ⟨a⟩ = (+ v) in id⟨[a]⟩ x
```

Observe que esta função seria normalmente escrita em termos de `map` do seguinte modo:

```
incBy x v = map (+ v) x
```

Observando as definições anteriores de `incBy` podemos perceber as seguintes diferenças entre `map` e `id` (utilizando redefinição local): 1) a função `map` pode ser utilizada apenas sobre listas, enquanto `id` pode ser utilizada sobre um tipo arbitrário; 2) `map` tem um tipo menos restrito. Se definirmos:

```
map'f = let id⟨a⟩ = f in id⟨[a]⟩
```

podemos notar que `map'` possui um tipo mais restrito que `map`:

```
map'  ::   ∀a::⋆.   (a→a)  →  [a]  →  [a]
 map  ::  ∀a,b::⋆.  (a→b)  →  [a]  →  [b]
```

A função passada como argumento para `map` pode alterar o tipo de seu argumento, ao contrário de `map'`, que deve preservar o tipo de seu argumento. Motivados por esta limitação, poderíamos tentar passar uma função de tipo `a → b` para a função `id`, em uma redefinição local. A função `id⟨[a]⟩` possui o seguinte tipo:

```
id⟨[a]⟩  ::  ∀a::⋆.  (id⟨a⟩ ::  a → a)  ⇒  (a→a)  →  [a]  →  [a],
```

Basta então renomear a função `id` para `map`, mantendo a mesma definição, para obter a seguinte anotação de tipo diferente:

```
map⟨[a]⟩  ::  ∀a::⋆, b::⋆.  (map⟨a⟩ ::  a → b)  ⇒  [a]  →  [b],
```

Observe que agora esta função está parametrizada sobre duas variáveis de tipo, assim como suas dependências. Esta definição de `map` pode ser utilizada para quaisquer tipos algébricos, como é ilustrado pelos exemplos a seguir:

```
map⟨[]⟩   (+ 1)          [1,2,3]
map⟨(,)⟩  (* 2) ("a"++)  (2, "bc")
```

A avaliação dessas expressões resulta, respectivamente, nos valores `[2,3,4]` e `(4, "abc")`. As expressões acima, são abreviações sintáticas para:

```
let map⟨a⟩ = (+ 1) in map⟨[a]⟩ [1,2,3]
```

e

```
let map⟨a⟩ = (* 2),
    map⟨b⟩ = ("a"++)
in map⟨(a,b)⟩ (2, "bc")
```

Apesar da definição de `map` em Generic Haskell ser simples, ela requer do programador conhecimento sobre o mecanismo de redefinição local (ou especialização). Note que essa definição de `map` é totalmente dependente deste recurso, uma vez que a aplicação de uma função a cada elemento de uma estrutura de dados é feita redefinindo o comportamento de `map` para o tipo dos elementos desta estrutura.

A.2.3 Sistema CT

A definição de `map`, em uma linguagem que utilize o Sistema CT, é simples e direta:

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

e podemos fornecer diversas definições, cada uma sobre um tipo diferente:

```
overload map f g (x, y) = (f x, g y)


overload map f  _ (Left x) = Left (f x)
         map _  g (Right x) = Right (g x)
```

As definições anteriores possuem os seguintes tipos:

```
map ::  (a → b) → [a] → [b]
map ::  (a → c) → (b → d) → (a,b) → (c,d)
map ::  (a → c) → (b → d) → Either a b → Either c d
```

O tipo inferido para `map` em um contexto em que essas definições são visíveis é:

```
map ::  {map :a→b→c} .a→b→c
```

Apesar da simplicidade das definições de `map`, no Sistema CT, não é possível definir a função `map` para tipos que possuem variáveis de *kind* superior, como o tipo `GRose`:

```
data GRose c a = GNode a (c (GRose c a))
```

Isto se deve ao fato de o Sistema CT não provê suporte a tipos de rank superior. Considere a seguinte tentativa de definição de `map` para o tipo `GRose`:

```
map f (GNode x y) = GNode (f x) (map f y)
```

Observando as aplicações da função `f` fornecida como parâmetro para `map`, podemos notar que esta é aplicada sobre dois tipos diferentes: em aplicação `(f x)`, a função `f` é aplicada a um valor de tipo `a`, enquanto em `(map f y)`, a função `f` é aplicada a um valor de tipo `c (GRose c a)`. Portanto, `f` é polimórfica e a função `map` para `GRose` é uma função polimórfica de rank 2.

Finalmente, observe que não é possível definir uma função `map` genérica no sistema CT e nem uma função `map` sobre tipos de dados não regulares ou sobre tipos que possuem variáveis de tipos de *kind* superior.

Referências Bibliográficas

[AM03] Thorsten Altenkirch and Conor McBride, *Generic programming within dependently typed programming*, Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming (Deventer, The Netherlands, The Netherlands), Kluwer, B.V., 2003, pp. 1–20.

[AP01] Artem Alimarine and Marinus J. Plasmeijer, *A generic programming extension for clean*, Implementation of Functional Languages, 2001, pp. 168–185.

[BP99] Richard S. Bird and Ross Paterson, *de bruijn notation as a nested datatype*, J. Funct. Program. **9** (1999), no. 1, 77–91.

[CaLFN07] Carlos Camarão, Cristiano Vasconcellos an Lucilia Figueiredo, and João Nicola, *Open and closed worlds for overloading: a definition and support for coexistence*, SBLP'2007 (submitted), 2007.

[CF99a] Carlos Camarao and Lucilia Figueiredo, *Type inference for overloading*, FLOPS'99 - International Workshop on Logic and Functional Programming, 1999.

[CF99b] ———, *Type inference for overloading without restrictions, declarations or annotations*, Fuji International Symposium on Functional and Logic Programming, 1999, pp. 37–52.

[CKJM05] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow, *Associated types with class*, POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), ACM Press, 2005, pp. 1–13.

[DM82] Luis Damas and Robin Milner, *Principal type-schemes for functional programs*, POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), ACM Press, 1982, pp. 207–212.

[DO02] Dominic Duggan and John Ophel, *Type-checking multi-parameter type classes*, J. Funct. Program. **12** (2002), no. 2, 133–158.

[dSOHL] Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Löh, *Generics as a library.*

[dW02] Jan de Wit, *A technical overview of generic haskell*, Master's thesis, Utrecht University, 2002.

[HHJW07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler, *A history of haskell: being lazy with class*, HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages (New York, NY, USA), ACM Press, 2007, pp. 12–1–12–55.

[Hin69] J. Roger Hindley, *The principal type-scheme of an object in combinatory logic*, Transactions of the American Mathematical Society (1969), no. 146, 29–60.

[Hin00a] Ralf Hinze, *A new approach to generic functional programming*, Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 2000.

[Hin00b] ———, *Polytypic values possess polykinded types*, Mathematics of Program Construction, 2000, pp. 2–27.

[Hin04] ———, *Generics for the masses*, SIGPLAN Not. **39** (2004), no. 9, 236–243.

[HJ00] Ralf Hinze and Simon Peyton Jones, *Derivable type classes*, Haskell Workshop 2000, September 2000.

[HJ02a] Ralf Hinze and Johan Jeuring, *Generic haskell: Applications*, Summer School on Generic Programming, 2002.

[HJ02b] ———, *Generic haskell: Practice and theory*, Summer School on Generic Programming, 2002.

[HJL06] R. Hinze, J. Jeuring, and A. Löh, *Comparing approaches to generic programming in Haskell*, Spring School on Datatype-Generic Programming, 2006.

[HL06] Ralf Hinze and Andres Löh, *"scrap your boilerplate"revolutions.*, MPC, 2006, pp. 180–208.

[HLB06] Ralf Hinze, Andres Löh, and Bruno, *"scrap your boilerplate"reloaded*, FLOPS'2006 - International Workshop on Logic and Functional Programming (2006).

[HM98] Graham Hutton and Erik Meijer, *Monadic Parsing in Haskell*, Journal of Functional Programming **8** (1998), no. 4, 437–444.

[Hug99] J. Hughes, *Restricted datatypes in haskell*, 1999.

[JBM98] C. B. Jay, G. Bellè, and E. Moggi, *Functorial ml*, Journal of Functional Programming **8** (1998), no. 6, 573–619.

[JJ97] Patrik Jansson and Johan Jeuring, *Polyp a polytypic programming language extension*, POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), ACM Press, 1997, pp. 470–482.

[JJM97] S. Jones, M. Jones, and E. Meijer, *Type classes: an exploration of the design space*, 1997.

[Jon93] Mark P. Jones, *A system of constructor classes: overloading and implicit higher-order polymorphism*, FPCA '93: Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark (New York, N.Y.), ACM Press, 1993, pp. 52–61.

[Jon99] ———, *Typing haskell in haskell*, Proceedings of the 1999 Haskell Workshop, October 1999.

[Jon00] ———, *Type classes with functional dependencies*, Lecture Notes in Computer Science **1782** (2000), 230–??

[Jon03] Simon Peyton Jones, *Haskell 98 language and libraries: The revised report*, 2003.

[JP06] Johan Jeuring and Rinus Plasmeijer, *Generic programming for software evolution*, Tech. Report UU-CS-2006-024, Institute of Information and Computing Sciences, Utrecht University, 2006.

[Kae88] Stefan Kaes, *Parametric overloading in polymorphic programming languages*, ESOP '88: Proceedings of the 2nd European Symposium on Programming (London, UK), Springer-Verlag, 1988, pp. 131–144.

[KATP02] P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer, *Gast: Generic automated software testing*, 2002.

[KW94] A. J. Kfoury and J. B. Wells, *A direct algorithm for type inference in the rank-2 fragment of the second-order &lgr;-calculus*, LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming (New York, NY, USA), ACM, 1994, pp. 196–207.

[LCJ03] A. Löh, D. Clarke, and J. Jeuring, *Dependency-style generic haskell*, 2003.

[LH06] Andres Löh and Ralf Hinze, *Open data types and open functions*, PPDP '06: Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming (New York, NY, USA), ACM, 2006, pp. 133–144.

[LJ95] John Launchbury and Simon Peyton Jones, *State in haskell*, Lisp Symb. Comput. **8** (1995), no. 4, 293–341.

[LJ03] Ralf Lämmel and Simon Peyton Jones, *Scrap your boilerplate: a practical design pattern for generic programming*, TLDI03: Proceedings of the ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, vol. 38, ACM Press, March 2003, pp. 26–37.

[LJ04] ———, *Scrap more boilerplate: reflection, zips, and generalised casts*, ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming (New York, NY, USA), ACM Press, 2004, pp. 244–255.

[Löh04] Andres Löh, *Exploring generic haskell*, Ph.D. thesis, Utrecht University, 2004.

[LP05] Ralf Lämmel and Simon Peyton Jones, *Scrap your boilerplate with class: extensible generic functions*, Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005), ACM Press, September 2005, pp. 204–215.

[Mey97] Bertrand Meyer, *Object-oriented software construction*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson, *Functional programming with bananas, lenses, envelopes and barbed wire*, Proceedings 5th ACM Conf. on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA, 26–30 Aug 1991 (J. Hughes, ed.), vol. 523, Springer-Verlag, Berlin, 1991, pp. 124–144.

[Mit96] John C. Mitchell, *Foundations of programming languages*, MIT Press, Cambridge, MA, USA, 1996.

[Oka98] Chris Okasaki, *Purely functional data structures*, Cambridge University Press, New York, NY, USA, 1998.

[Oka99] ———, *From fast exponentiation to square matrices: an adventure in types*, ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming (New York, NY, USA), ACM Press, 1999, pp. 28–35.

[Pow] A.L. Powell, *A literature review on the quantification of software change*.

[PvENS] M.J. Plasmeijer, M.C.J.D. van Eekelen, E.G.J.M.H. Nöcker, and J.E.W. Smetsers, *The concurrent clean system - functional programming on the macintosh*.

[PVWS07] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields, *Practical type inference for arbitrary-rank types*, J. Funct. Program. **17** (2007), no. 1, 1–82.

[RH02] Andres Löh Ralf Hinze, Johan Jeuring, *Type-indexed data types*, Proceedings of the Sixth International Conference on Mathematics of Program Construction (MPC 2002), July 2002.

[Str00a] Christopher Strachey, *Fundamental concepts in programming languages*, Higher Order Symbol. Comput. **13** (2000), no. 1-2, 11–49.

[Str00b] Bjarne Stroustrup, *The c++ programming language*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[Tol] Andrew Tolmach, *An external representation for the GHC core language*.

[Vas04] Cristiano Damiani Vasconcelos, *Inferência de tipos com suporte a sobrecarga baseada no sistema ct*, 2004.

[Wad89] Philip Wadler, *Theorems for free!*, Proceedings 4th Int. Conf. on Funct. Prog. Languages and Computer Arch., FPCA'89, London, UK, 11–13 Sept 1989, ACM Press, New York, 1989, pp. 347–359.

[Wad92] ———, *Monads for functional programming*, Marktoberdorf Summer School on Program Design Calculi, August 1992.

[Wat90] David A. Watt, *Programming language concepts and paradigms*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[WB89] Philip Wadler and S. Blott, *How to make ad-hoc polymorphism less ad-hoc*, Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages, ACM, 1989, pp. 60–76.

[Wel94] J. B. Wells, *Typability and type checking in system f are equivalent and undecidable*, Proceedings of the Ninth Annual IEEE Symp. on Logic in Computer Science, LICS 1994 (Samson Abramsky, ed.), IEEE Computer Society Press, July 1994, pp. 176–185.