FUNÇÕES SOBRECARREGADAS COMO OBJETOS DE PRIMEIRA CLASSE

ELTON MÁXIMO CARDOSO

FUNÇÕES SOBRECARREGADAS COMO OBJETOS DE PRIMEIRA CLASSE

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Carlos Camarão de Figueiredo
Co-Orientador: Lucília Camarão de Figueiredo

Belo Horizonte

Janeiro de 2011

Cardoso, Elton Máximo
C268f Funções sobrecarregadas como objetos de primeira classe / Elton Máximo Cardoso. — Belo Horizonte, 2011
xii, 67 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais
Orientador: Carlos Camarão de Figueiredo

Co-Orientador: Lucília Camarão de Figueiredo

1. Computação. 2. Tese. 3. Linguagem de Programação (computadores) - Tese. I. Orientador II. Co-Orientador. III. Título.

CDU 519.6*33 (043)

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

FOLHA DE APROVAÇÃO

Funções sobrecarregadas como objeto de primeira classe

ELTON MAXIMO CARDOSO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

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

PROFA. LUCÍLIA CAMARÃO DE FIGUEIREDO - Co-orientadora
Departamento de Computação - UFOP

PROF. ATZE DJISKTRA
Instituto de Informação e Ciência da Computação - Universiteit de Utrecht

PROFA. MARIZA ANDRADE DA SILVA BIGONHA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 19 de novembro de 2010.

Agradecimentos

Ao Departamento de Computação da Universidade Federal de Minas Gerais, pelas instalações, infraestrutura e apoio para o desenvolvimento deste trabalho. Aos professores Carlos Camarão Figueiredo e Lucília Camarão Figueiredo, respectivamente orientador e coorientador deste trabalho, pelo constante apoio, estímulo e pelo rigor nas inúmeras correções que se fizeram necessárias ao longo do trabalho. Aos professores das disciplinas que cursei, pelas valiosas lições. Agradeço aos meus pais pela compreensão, e pelo amparo durante o tempo em que esse trabalho foi realizado. Ao meu irmão, por seus bons conselhos e por ser sempre tão prestativo. Agradeço ao amigo Rodrigo Geraldo Ribeiro por ter cedido o código utilizado como base para implementação descrita neste trabalho e também pela companhia e pelos bons temas de discussão durante o café. Aos amigos Leonardo V. S. R., Sérgio M. D. e Henrique L. B. pelo companheirismo. Aos amigos Euler H. M. e Pedro C. G. pela assistência em Linux nas horas de desespero.

"Por natureza, o egoísmo é ilimitado: o homem quer conservar a sua existência utilizando qualquer meio ao seu alcance, quer ficar totalmente livre das dores que também incluem a falta e a privação, quer a maior quantidade possível de bem-estar e todo o prazer de que for capaz, e chega até mesmo a tentar desenvolver em si mesmo, quando possível, novas capacidades de deleite. Tudo o que se opõe ao ímpeto do seu egoísmo provoca o seu mau humor, a sua ira e o seu ódio: ele tentará aniquilá-lo como a um inimigo. Quer possivelmente desfrutar de tudo e possuir tudo; mas, como isso é impossível, quer, pelo menos, dominar tudo: "Tudo para mim e nada para os outros"é o seu lema. O egoísmo é gigantesco: ele rege o mundo."

(Arthur Schopenhauer - A Arte de Insultar)

Resumo

O uso de polimorfismo em linguagens de programação constitui um importante recurso para reuso de código e para clareza e concisão de programas. A base para implementação de polimorfismo em linguagens de programação modernas é o sistema de tipos de Hindley-Milner (HM) e uma extensão bastante útil desse sistema é a possibilidade de definição de símbolos sobrecarregados (com tipo polimórfico restrito). No sistema HM a inferência de tipos é relativamente simples, em razão de sua restrição de que parâmetros de funções devem ter tipo monomórfico. Esse requerimento pode entretanto ser um inconveniente em diversas aplicações. Este trabalho propõe uma extensão ao sistema de tipos de Hindley-Milner + sobrecarga, a qual possibilita a definição de funções com parâmetros de tipo polimórfico, mediante anotação explícita do tipo de tais funções pelo programador, sendo esses tipos especificados na forma de tipos interseção. No sistema proposto, funções com parâmetros polimórficos podem tanto ser aplicadas a argumentos de tipo polimórfico paramétrico (como em sistemas de polimorfismo de rank superior), como a argumentos de tipo polimórfico restrito, promovendo valores sobrecarregados a objetos de primeira classe .

Abstract

The use of polymorphic abstractions in programmming languages constitutes an important tool for code reuse and for program clarity and conciseness. The basis of most modern languages for the exploitation of polymorphism is Hindley-Milner's type system, which has achieved such success due in great part to the relative simplicity of it's type inference mechanism. This simplicity is obtained, however, by imposing some restrictions. A major restriction is the (so-called) *no polymorphic abstraction*: function parameters must have a monomorphic type (in other words, parameters cannot be used with distinct types inside the function's body).

There has been much work on extensions to overcome this restriction, involving so-called higher rank (also called *rank-n*) type systems, which allow arguments of polymorphic type. This work follows such vein, but is distinguished from all previous related work as follows.

Intersection types are used to allow parameters to be used polymorphically inside a function's definition (the type of such polymorphic parameters must be explicitly annotated); this allows such function to receive arguments not only of polymorphic type may also of constrained polymorphic type (i.e. arguments involving the use of overloaded symbols). This promotes overloaded values to first-class.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Tipos têm papel cada vez mais importante na construção de programas, na engenharia de software e no raciocínio sobre propriedades de programas. Eles servem como documentação e especificação parcial da funcionalidade de programas e possibilitam garantir que determinadas classes de erros nunca irão ocorrer quando o programa for executado, podendo também contribuir para geração de código mais eficiente.

Sistemas de tipos para linguagens de programação consistem em métodos *sintáticos* para provar a ausência de determinados comportamentos indesejados de programas, por meio da classificação das frases do programa de acordo com os tipos de valores que elas computam. Esses comportamentos indesejados que podem ser eliminados por meio de sistemas de tipos são comumente denominados *erros de tipo*.

Sistemas de tipos são necessariamente *conservativos*, ou seja, para garantir que programas bem tipados não estão sujeitos a erros de tipo, é necessário algumas vezes rejeitar programas cujo comportamento, em tempo de execução, seria, de fato, correto. Por exemplo, um programa tal como

```
if <teste complexo> then 5 else <erro de tipo>
```

é rejeitado como não corretamente tipado, mesmo no caso em que a avaliação de `<teste complexo>` sempre resulte `True`, já que, em geral, não é possível determinar que é esse o caso por meio de análise puramente sintática.

Pesquisas em sistemas de tipos para linguagens de programação buscam atribuir tipos cada vez mais informativos a expressões de programas, de maneira a eliminar o maior número possível de comportamentos indesejados e ao mesmo tempo permitir que maior número de programas com comportamento desejado sejam considerados válidos, ou bem tipados.

Um passo importante no sentido de obter uma disciplina de tipos que evite o uso de expressões em contextos inapropriados, levando em conta, ao mesmo tempo, o objetivo de possibilitar reuso de código, foi a definição de sistemas de tipos *polimórficos*. Um tipo polimórfico consiste em um *esquema* (conjunto, coleção) de tipos, envolvendo variáveis de tipo universalmente quantificadas, e expressa que uma determinada funcionalidade é oferecida para *todas* as instâncias desse esquema de tipos. Por exemplo, o tipo da função de aplicação — que toma dois argumentos e aplica o primeiro argumento ao segundo — pode ser escrito como $\forall\, a\, b.\, (a \to b) \to a \to b$. Esse tipo expressa, de fato, toda uma coleção de tipos, obtidos pela instanciação de cada uma das variáveis universalmente quantificadas para um tipo específico arbitrário.

A forma mais simples de *polimorfismo paramétrico* foi definida no sistema de tipos proposto independentemente por Hindley [16] e por Damas-Milner [10]. No sistema de Hindley-Milner (`HM`), quantificadores ocorrem apenas nas posições mais externas de esquemas de tipos: parâmetros de funções não podem ter tipo polimórfico.

Toda expressão bem tipada no sistema `HM` pode ser caracterizada por um *tipo principal* — um tipo cujo conjunto de instâncias é constituído exatamente por todos os tipos que podem ser atribuídos a essa expressão. Além disso, o tipo principal de cada expressão pode ser inferido automaticamente, sem que seja requerida qualquer anotação de tipo em programas.

O sistema `HM` tem sido usado como base para os sistemas de tipos de linguagens funcionais modernas, tais como ML [34] e Haskell [39], e as vantagens do uso de polimorfismo paramétrico têm sido cada vez mais reconhecidas, como mostra sua introdução também em linguagens como C# e Java versão 5. O sucesso do sistema `HM` é devido à sua expressividade e ao fato de possuir um algoritmo de inferência de tipos relativamente simples.

Outra forma importante de polimorfismo é o chamado *polimorfismo restrito*, ou *polimorfismo de sobrecarga* [46], no qual uma determinada funcionalidade é provida para diferentes tipos, com diferentes implementações para cada tipo. Por exemplo, um teste de igualdade para uma lista de inteiros é implementado de forma diferente do teste de igualdade para inteiros, ou para árvores, ou para valores booleanos. Em Haskell, o mecanismo de sobrecarga é suportado por meio da definição de *type classes*, ou classes de tipos, originalmente proposto em [53]. Por exemplo, uma classe `Eq`, com operadores (`==`) e (`/=`) para igualdade e desigualdade, respectivamente, pode ser declarada do seguinte modo:

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

Os operadores `(==)` e `(/=)` têm tipo polimórfico restrito: $\forall a.\,\texttt{Eq}\ a \Rightarrow a \to a \to \texttt{Bool}$. A restrição de tipo (ou restrição de classe) `Eq` a indica que a variável a pode ser instanciada apenas para tipos que sejam instâncias da classe `Eq`. Uma definição de *instância de classe* provê implementações apropriadas para os símbolos declarados nessa classe. Por exemplo, tipos básicos, como `Int`, podem ser definidos como instâncias da classe `Eq`, sendo os testes de igualdade e desigualdade implementados por funções pré-definidas. Outros tipos também podem ser definidos como instâncias da classe `Eq`, tal como no seguinte exemplo, que define uma instância para listas:

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

Nesse exemplo, o teste de igualdade para elementos de um tipo arbitrário, mas fixo, é usado para definir a igualdade para listas de elementos desse tipo. Por meio dessa definição, o teste de igualdade pode agora ser usado para valores de tipo `[Int]`, `[[Int]]`, e assim por diante. E o mesmo vale para outras funções, definidas em termos do operador de igualdade, tais como:

```
elem ::  ∀a. Eq a ⇒ a → a → Bool
elem y = foldr (λ x r → (x == y) || r) False
```

A clara fundamentação matemática de linguagens de programação funcional faz com que sejam particularmente apropriadas para a formalização, análise e experimentação de sistemas de tipos cada vez mais expressivos. Haskell, em particular, tem sido usada nas duas últimas décadas como um laboratório para explorar, projetar e implementar idéias novas e avançadas relacionadas a sistemas de tipos [17]. Um problema particularmente importante, que tem sido objeto de diversos trabalhos de pesquisa recentes, é a extensão do sistema de tipos de Hindley-Milner de maneira a possibilitar a definição de funções com parâmetros de tipos polimórficos. Por exemplo, considere a seguinte definição de função:

Exemplo 1.1 `foo g = (g [True,False], g ['b','a','c'])`

Note que, no corpo da função `foo`, o parâmetro `g` é aplicado a listas de dois diferentes tipos. Gostaríamos que o tipo atribuído a `foo` fosse tal que possibilitasse, por exemplo, a aplicação de `foo` à função polimórfica `length`, que computa o número de elementos de uma lista, dando como resultado, nesse caso, o valor `(2,3)`. Mas tal aplicação não seria bem tipada no sistema de Hindley-Milner. Nesse sistema, poderíamos atribuir a `foo` o tipo $\forall a.(\texttt{[}a\texttt{]} \rightarrow \texttt{Int}) \rightarrow (\texttt{Int}, \texttt{Int})$, que entretanto não seria satisfatório. Esse tipo significa que `foo` é uma função polimórfica que, para qualquer tipo específico, digamos, τ, toma como argumento uma função de tipo $\texttt{[}\tau\texttt{]} \rightarrow \texttt{Int}$ e produz como resultado um valor de tipo (`Int`,`Int`). Mas, nesse caso, o tipo τ não poderia ser tanto `[Bool]` quanto `[Char]`, como é requerido pelos usos do parâmetro (`g`) no corpo da função `foo` (isto é, nas aplicações `g [True,False]` e `g ['b','a','c']`, respectivamente). A questão aqui é que desejamos expressar que `foo` deve poder ser aplicada a um argumento de tipo polimórfico, que possa ser instanciado para os tipos $\texttt{[Bool]} \rightarrow \texttt{Int}$ e $\texttt{[Char]} \rightarrow \texttt{Int}$.

Em um sistema mais expressivo, o tipo de `foo` poderia ser $(\forall a.\texttt{[}a\texttt{]} \rightarrow \texttt{Int}) \rightarrow (\texttt{Int}, \texttt{Int})$. Esse tipo não é válido no sistema `HM`, pois envolve quantificadores que aparecem internamente ao tipo, nesse caso do lado esquerdo do construtor de tipo funcional. De modo geral, podem ser necessários tipos em que quantificadores possam ocorrer em qualquer posição. Essa possibilidade de tratar tipos polimórficos como qualquer outro tipo é chamada de *polimorfismo de primeira classe* (ou *polimorfismo de rank superior*, ou de *rank-n*).

Existem diversas aplicações interessantes que requerem polimorfismo de rank-n, apresentadas, por exemplo, em [45, 38, 50], incluindo abstrações sobre mônadas, definições de invariantes de tipos de dados e funções genéricas (ou *politípicas*). Isso motivou a introdução de polimorfismo de *rank-n* em algumas implementações correntes da linguagem Haskell [38].

O sistema `F`, de Girard-Reynolds [14, 42], também chamado lambda-calculus polimórfico, ou lambda-caulculus de segunda ordem, constitui a referência fundamental para polimorfismo de rank-n. Ao contrário do que ocorre no sistema de Hindley-Milner, a inferência de tipos no sistema `F` é indecidível [55]. Além disso, no sistema `F` nem toda expressão possui tipo principal.

Para contornar esses problemas, diversos trabalhos recentes buscam definir um sistema de tipos que suporte polimorfismo de rank-n, mas preserve a propriedade de tipo principal e a decidibilidade da inferência de tipos, para isso requerendo anotações explícitas de tipos em programas. Trabalhos relevantes nesse sentido são `MLF` [28, 29],

FPH [52] e Flexible Types [30, 31].

O uso de sistemas de tipos com suporte a polimorfismo de rank-n certamente introduz flexibilidade e expressividade adicional na linguagem. Entretanto, em alguns casos tipos polimórficos não capturam de maneira adequada as restrições que se deseja expressar em relação a parâmetros de funções.

Considere os seguintes usos da função foo definida no exemplo 1.1 `(foo id)`, `(foo const)`, `(foo reverse)`, `(foo head)`, `(foo sort)`, `(foo allEqual)`, `(foo fromEnum)`, `(foo blah)` onde

`id` :: $\forall a.\, a \to a$	$--$ função identidade
`consti` :: $\forall a.\, a \to$ `Int`	$--$ função constante
`reverse` :: $\forall a.\, [a] \to [a]$	$--$ inversão de lista
`head` :: $\forall a.\, [a] \to a$	$--$ primeiro elemento de lista
`sort` :: $\forall a.$ `(Ord` a`)` $\Rightarrow [a] \to [a]$	$--$ ordenação
`allEqual` :: $\forall a.$ `(Eq` a`)` $\Rightarrow [a] \to$ `Bool`	$--$ todos os elementos iguais
`fromEnum` :: $\forall a.$ `(Enum` a`)` $\Rightarrow a \to$ `Int`	$--$ enumeração para inteiro
`blah` :: $\forall a, b.$ `(C` a b`)` $\Rightarrow [a] \to [b]$	

Os quatro primeiros argumentos são funções polimórficas paramétricas e os quatro últimos funções com tipo polimórfico restrito por *predicados de classe*, o que significa que tais funções são definidas por expressões que envolvem símbolos sobrecarregados.

Uma possível anotação de tipo para `foo` seria `foo` :: $\forall c\, d.\, (\forall a\, b.\, a \to b) \to (c, d)$, mas esse tipo permitiria a aplicação de `foo` apenas à função `undefined`. Se a anotação de tipo para `foo` :: $\forall b.\, (\forall a.\, a \to b) \to (b, b)$, a única dentre as aplicações acima que seria bem tipada seria `(foo const)` (com tipo `(Int,Int)`). Para que a aplicação `(foo id)` seja bem tipada, a anotação de tipo deve ser `foo` :: $(\forall a.\, a \to a) \to$ (`[Bool]`, `[Char]`) e, nesse caso, os demais usos de `foo` relacionados acima não seriam válidos.

A aplicação `(foo sort)` pode ser permitida em um sistema de tipos tal como `QMLF` [33]. Esse sistema de tipos estende o sistema `MLF` de maneira a permitir tipos polimórficos de rank-n *com restrições*. Como observa o próprio autor, a maior dificuldade para prover essa extensão não reside propriamente nas modificações requeridas nas regras do sistema de tipos `MLF` ou no seu mecanismo de inferência de tipos, mas sim na implementação do sistema resultante, que deve lidar corretamente com o mecanismo usual de implementação de funções sobrecarregadas, baseado no uso de *dicionários* [53].

Em implementações de Haskell com tal extensão, a aplicação `(foo sort)` é bem

tipada se a anotação de tipo provida para `foo` for, por exemplo

$$\texttt{foo} :: (\forall a.\, \texttt{(Ord}\ a\texttt{)} \Rightarrow [a] \rightarrow [a]) \rightarrow (\texttt{[Bool]}, \texttt{[Char]})$$

sendo nesse caso também bem tipada a aplicação `(foo id)`, já que o tipo de `id` pode ser instanciado para o tipo do parâmentro de `foo`, no sistema `QMLF`. Entretanto, essa anotação de tipo não possibilitaria a aplicação `(foo buzz)`, onde `buzz::`$(\forall a.\, \texttt{(C}\ a\texttt{)} \Rightarrow [a] \rightarrow [a]) \rightarrow (\texttt{[Bool]}, \texttt{[Char]})$ é uma função com tipo polimórfico quase idêntico ao de `sort`, mas porém com uma restrição de classe `(C` a`)`, em que `C` é uma classe distinta de `Ord`.

Neste trabalho, buscamos definir um sistema de tipos no qual seja possível expressar de maneira mais adequada e precisa o tipo de uma função com parâmetro polimórfico, tal como a função `foo` do exemplo acima, possibilitando o uso dessa função, por exemplo, em qualquer das expressões consideradas anteriormente. Ou seja, o tipo de `foo` deve expressar adequadamente que o seu parâmetro é uma função que pode ser aplicada tanto a um argumento de tipo `[Bool]` como a um argumento de tipo `[Char]` e, além disso, que o primeiro e o segundo componentes do par resultante devem ter tipo, respectivamente, igual ao do valor resultante da aplicação do argumento de `foo` a uma expressão de tipo `[Bool]` e a uma expressão de tipo `[Char]`.

Para expressar esses requerimentos de maneira adequada, o sistema de tipos proposto utiliza uma forma restrita de *tipos interseção* [8]. Nesse sistema, o tipo da função `foo` do exemplo 1.1 poderia ser anotado como

$$\texttt{foo} :: \forall a\, b.\, (\texttt{[Bool]} \rightarrow a \wedge \texttt{[Char]} \rightarrow b) \rightarrow (a, b)$$

Mediante essa anotação de tipo na definição de `foo`, qualquer das aplicações anteriormente seria válida. Por exemplo, `(foo id)` :: (`[Bool]`, `[Char]`), `(foo consti)` :: (`Int`, `Int`) e `(foo blah)` :: $\forall a\, b.\, (\texttt{C Int}\ a, \texttt{C Bool}\ b) \rightarrow (a, b)$. Nesse último caso, a sobrecarga pode ainda ser resolvida, de acordo com o contexto em que essa expressão for utilizada.

As contribuições dessa dissertação são:

1. A definição de um sistema de tipos original, que denominamos `CTi`, que provê suporte a passagem de funções sobrecarregadas para funções de *rank* superior, isto é, funções usadas polimorficamente na sua própria definição.

2. Uma descrição (informal) da semântica para termos da linguagem do sistema `CTi`.

3. A definição de um algoritmo para inferência de tipos para o sistema `CTi`.

4. Uma implementação desse algoritmo de inferência de tipos, em Haskell, para inferência de tipos para expressões de uma linguagem de programação que consiste em uma extensão de Haskell para suporte ao sistema de tipos proposto.

5. A elaboração de uma revisão sobre sistemas de tipos para liguagens de programação, que aborda o sistema de tipos polimórficos de *let-bound* de Hindley-Milner, a extensão desse sistema para suporte a sobrecarga e sistemas com suporte para funções de rank superior. Essa revisão, embora não seja original, contribui para difundir e disseminar o assunto, particularmente para leitores ainda não muito experientes.

Um dos objetivos do sistema de tipos aqui proposto, é servir como base para a definição de um sistema ainda mais flexível, que possibilite a passagem de funções sobrecarregadas g como argumento para uma função f, de maneira que, no corpo de f, g possa ser aplicada a estruturas de dados com elementos de tipos heterogêneos, tais como uma lista heterogênea. A idéia é definir o tipo dessa lista heterogênea como um tipo $[T]$, em que T representa todos os possíveis tipos de elementos dessa lista; uma lista com tal tipo poderia ser usada seguramente (sem possibilidade de erros de tipo) como argumento de uma função sobrecarregada g, se, para cada um dos tipos representados por T, existe uma definição de uma instância em que o parâmetro de g tem esse tipo.

1.1 Organização

O restante do texto está organizado da seguinte forma. O Capítulo 2 apresenta uma revisão do sistema de tipos polimórficos de rank-1 proposto por Hindley e Milner (`HM`) e uma revisão sucinta sobre sistemas de tipos polimórficos de rank superior, incluindo o sistema `F`, proposto por Girard e Reynolds. Este Capítulo também discute brevemente alguns trabalhos que propõem extensões de `HM` para suporte a polimorfismo de rank superior.

O Capítulo 3 apresenta uma extensão do sistema `HM` para suporte a sobrecarga, baseada no sistema de classes de tipos de Haskell, a qual constitui a base para o sistema proposto nesse trabalho. Os dois Capítulos também introduzem também os conceitos e notação relativas a sistemas de tipos usados na paresentação do sistema de tipos proposto.

O Capítulo 4 apresenta as regras do sistema de tipos `CTi`, proposto neste trabalho, assim como uma semântica para expressões desse sistema e uma definição do algoritmo de inferência de tipos correspondente.

Finalmente, no Capítulo 5 são apresentadas as conclusões do trabalho.

Capítulo 2

Polimorfismo Paramétrico

Este capítulo provê uma revisão sobre sistemas de tipos com suporte a polimorfismo paramétrico. A Seção 2.1 introduz a linguagem de tipos e expressões usada como base para a definição desses sistemas de tipos e define formalmente a noção de *rank* de tipos polimórficos.

A Seção 2.2 aborda polimorfismo de *rank*-1, apresentando o sistema de tipos de Hindley-Milner[16, 10], assim como a inferência de tipos para esse sistema e a semântica de termos da sua linguagem. O sistema `HM`, usado como base para linguagens funcionais modernas, tem a importante propriedade de que o tipo de qualquer expressão da linguagem pode ser inferido automaticamente, sem que qualquer anotação explícita de tipo seja requerida em programas. Entretanto, quase toda extensão desse sistema, como por exemplo a introdução de polimorfismo de *rank* superior, destrói essa propriedade, requerendo a introdução de anotações de tipo em determinados pontos do programa, de modo a garantir decidibilidade da inferência de tipos.

A Seção 2.3 discute polimorfismo de *rank* superior, apresentando o sistema `F` [18], que constitui a referência básica para sistemas de polimorfismo de *rank* arbitrário. Na Seção 2.3.2 discutimos sucintamente alguns trabalhos recentes que buscam definir sistemas de tipo de ordem superior preservando a decidibiliade da inferência de tipos, por meio do requerimento de anotações de tipo em determinados pontos do programa, e mantendo compatibilidade com o sistema `HM`, na ausência de anotações de tipo.

2.1 Conceitos Básicos

A linguagem básica de tipos a ser utilizada nos sistemas de tipos discutidos neste capítulo é apresentada na Figura 2.1; Extensões dessa linguagem de tipos introduzidas oportunamente.

Hyndley-Milner	sistema F
rank 1	*rank* arbitrário
$\rho ::= \tau$	$\rho ::= \tau \mid \sigma \to \sigma$

Variáveis de tipo	a, b, c		
Tipos monomórficos	τ	$::=$	$a \mid \tau_1 \to \tau_2$
Tipos polimórficos	σ	$::=$	$\forall \overline{a}.\rho$
Contexto de tipos	Γ	$::=$	$\emptyset \mid \Gamma, x : \sigma$

Figura 2.1. Sintaxe de tipos

Tipos monomórficos (τ) consistem de tipos simples, representados por variáveis de tipo (a, b), e tipos funcionais (o construtor de tipos funcionais ($\to$) é escrito de forma infixada, como usual). Outros construtores de tipo comumente usados em linguagens de programação podem também ser incluídos, se desejado.

Tipos polimórficos (σ) incluem quantificação universal sobre variáveis de tipo. A notação $\overline{a}$ é usada nessa Figura, assim como no restante do texto, para representar uma seqüência de zero ou mais variáveis de tipo $a_1, \ldots, a_n$. A notação $\forall \overline{a}.\rho$ não necessariamente representa quantificação sobre todas as variáveis de tipo livres de ρ, ou seja, um tipo polimórfico pode ter variáveis de tipo livres. Dizemos que um tipo é *fechado* se ele não possui variáveis livres.

A Figura 2.1 introduz também a definição de um contexto de tipos (Γ), que provê informação sobre os tipos de variáveis visíveis em um dado escopo. A associação de um tipo σ a uma variável x no contexto de tipos é denotada por $x : \sigma$. A notação $\Gamma(x)$ denota o tipo associado à variável x no contexto Γ e Γ_x denota o contexto obtido removendo-se de Γ a atribuição de tipo a x que eventualmente ocorra nesse contexto. Definimos $\Gamma, (x : \sigma) = \Gamma_x \cup \{(x : \sigma)\}$ e

$$\Gamma'_{[x,\Gamma]} = \begin{cases} \Gamma'_x, (x : \sigma) & \text{se } (x : \sigma) \in \Gamma \\ \Gamma'_x & \text{caso contrário} \end{cases}$$

O conjunto de variáveis livres de um tipo σ, $ftv(\sigma)$, é definido da maneira usual, sendo essa definição também estendida para contextos de tipos da maneira usual, ou seja, $ftv(\Gamma) = \bigcup\{ftv(\sigma) \mid (x : \sigma) \in \Gamma\}$. Escrevemos abreviadamente $ftv(o, o')$ para denotar o conjunto de variáveis $ftv(o) \cup ftv(o')$, para objetos o e o'.

A notação $[\overline{a \mapsto \rho}]\sigma$ denota o tipo obtido pela substituição simultânea, em σ, de

Termos	t, u	$::=$	x	Variável
		$\mid$	$\lambda x.\, t$	Abstração funcional
		$\mid$	$t\ u$	Aplicação
		$\mid$	$\texttt{let}\ x = u\ \texttt{in}\ \ t$	Definição local
		$\mid$	$t :: \sigma$	Anotação de tipo (σ fechado)

Figura 2.2. Sintaxe do lambda-calculus

todas as ocorrências livres de cada uma das variáveis $a_1, \ldots, a_n \in \overline{a}$, $n \geq 0$, pelo tipo correspondente $\rho_1, \ldots, \rho_n \in \overline{\rho}$, sendo essa substituição válida apenas se não resulta em captura de variáveis livres em σ. [1] Essa substituição é também escrita na forma $S\sigma$, onde S é uma função de variáveis de tipo em tipos ρ e $S(a_i) = \rho_i$, para $i \geq 0$ e $S(b) = b$, se $b \notin \overline{a}$. A composição de duas substituições R e S é definida da maneira usual e escrita como $R \circ S$.

Os sistemas de tipos a serem considerados neste trabalho diferem na sua definição dos tipos intermediários ρ, cuja característica é não apresentar quantificadores no nível mais externo. No sistema de tipos de Hyndley-Milner, quantificadores ocorrem apenas no nível mais externo, restringindo o sistema a polimorfismo de rank-1. O sistema `F` inclui tipos polimórficos de rank arbitrário. O *rank* de um tipo descreve a profundidade na qual quantificadores universais ocorrem em posição contravariante no tipo [26]:

$$\begin{array}{llll} \text{Tipos monomórficos} & \tau, \sigma^0 & ::= & a \mid \tau_1 \to \tau_2 \\ \text{Tipos polimórficos} & \sigma^{n+1} & ::= & \sigma^n \mid \sigma^n \to \sigma^{n+1} \mid \forall a.\sigma^{n+1} \end{array}$$

Por exemplo:

$\texttt{Int} \to \texttt{Int}$	Rank 0
$\forall a.a \to a$	Rank 1
$\texttt{Int} \to (\forall a.a \to a)$	Rank 1
$(\forall a.a \to a) \to \texttt{Int}$	Rank 2

A sintaxe da linguagem de termos (ou expressões) dos sistemas de tipos discutidos nesse trabalho é apresentada na Figura 2.2. A linguagem consiste do *lambda-calculus* [2] estendido com definições locais não recursivas (*let-bindings*) e com a possibilidade de incluir anotações de tipos explícitas. Definições locais recursivas são omitidas uma vez que não introduzem dificuldade adicional do ponto de vista da inferência de tipos.

[1] Veja [40] para uma definição precisa das noções de variáveis livres e de substituição sem captura de variáveis.

2.2 Polimorfismo de *Rank*-1

A possibilidade de inclusão de anotações de tipo explícitas, escritas usando "::", é considerada pelas seguintes razões: 1) para mostrar como anotações de tipo podem ser usadas para dirigir a inferência de tipos, de uma maneira simples e previsível; 2) no sistema de tipos proposto nesse trabalho, anotações de tipo serão usadas para possibilitar inferir tipos que não seriam inferidos no sistema `HM`, sendo mantida total compatibilidade com esse sistema na ausência de anotações de tipo. Supomos, por simplicidade, que os tipos especificados em anotações de tipo são *fechados*, isto é, não incluem variáveis de tipo livres.

O sistema de tipos de Hindley-Milner[16] (também chamado de Damas-Milner[10]) foi um dos primeiros sistemas de tipos a incorporar uma forma simples de polimorfismo (*let-bound*) que possibilita a definição de funções polimórficas de rank-1. Sistemas de tipos de linguagens de programação funcionais modernas, tais como Haskell[39] e ML[34], e até mesmo linguagens imperativas, como Java (a partir de sua versão 1.5, com a introdução de *classes genéricas*), baseiam-se na idéia de polimorfismo introduzida por este sistema de tipos. A utilidade de polimorfismo paramétrico em linguagens de progtramação é largamente reconhecida, e pode ser ilustrada por meio dos exemplos a seguir, escritos na linguagem Haskell.

O primeiro exemplo apresenta a definição de uma função `length`, que retorna o comprimento de uma lista. Note que o tipo da função é polimórfico, o que possibilita que essa função possa ser aplicada a listas de elementos de qualquer tipo, evitando que se tenha que prover uma nova definição para essa função para listas de cada tipo sobre a qual se deseja aplicar essa operação, como seria o caso em um sistema de tipos monomórfico.

```
length ::  [a] -> Int
length []     = []
length (x:xs) = 1 + length xs
```

Outro exemplo é o da função `map`, que captura o padrão de computação da operação de aplicar uma determinada operação a todos os elementos de uma lista. Essa função pode ser definida (recursivamente) do seguinte modo:

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

Note que, como `map` tem tipo polimórfico, pode ser aplicada a listas de elementos

de diferentes tipos, tal como, por exemplo, nas expressões:

```
map (+1) [1, 2, 3]    = [2, 3, 4]
map not [True, False] = [False, True]
map even [1, 2, 3]    = [False, True, False]
```

Polimorfismo paramétrico pode também ser usado na definição de tipos algébricos, tal como no exemplo a seguir, em que é definido o tipo polimórfico `Tree a`, que representa árvores binárias que armazenam nas folhas valores de um tipo a arbitrário.

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

2.2.1 Sistema de Tipos

O sistema de tipos `HM` é apresentado na Figura 2.3, na forma de um conjunto de regras para a derivação de um julgamento da forma

$$\Gamma \vdash t : \sigma$$

que especifica que "o termo t tem tipo σ no contexto de tipos Γ". Note entretanto que várias regras incluem julgamentos da forma $\Gamma \vdash t : \rho$, como, por exemplo, a regra `(APP)`, significando, nesse caso, que o tipo não possui quantificadores.

A regra `(VAR)` especifica que o tipo de uma variável é determinado pela informação

$$\rho ::= \tau$$

$$\boxed{\Gamma \vdash t : \sigma}$$

$$\frac{}{\Gamma, (x : \sigma) \vdash x : \sigma}\ \texttt{(VAR)}$$

$$\frac{\Gamma,\ (x : \tau) \vdash t : \rho}{\Gamma \vdash \lambda x.\, t : \tau \to \rho}\ \texttt{(ABS)} \qquad \frac{\Gamma \vdash t : \tau \to \rho \quad \Gamma \vdash u : \tau}{\Gamma \vdash t\ u : \rho}\ \texttt{(APP)}$$

$$\frac{\Gamma \vdash u : \sigma \quad \Gamma, (x : \sigma) \vdash t : \rho}{\Gamma \vdash \texttt{let}\ \ x = u\ \texttt{in}\ \ t : \rho}\ \texttt{(LET)} \qquad \frac{\Gamma \vdash t : \sigma}{\Gamma \vdash (t :: \sigma) : \sigma}\ \texttt{(ANNOT)}$$

$$\frac{\Gamma \vdash t : \sigma \quad \overline{a} \notin ftv(\Gamma)}{\Gamma \vdash t : \forall \overline{a}.\, \rho}\ \texttt{(GEN)} \qquad \frac{\Gamma \vdash e : \forall \overline{a}.\, \rho}{\Gamma \vdash t : [\overline{a \mapsto \tau}]\rho}\ \texttt{(INST)}$$

Figura 2.3. Sistema de Tipos `HM`

de tipo correspondente disponível no contexto Γ. A regra (APP) determina que uma aplicação é bem tipada apenas se o tipo do argumento é igual ao do parâmetro da função. A regra (ABS) estabelece que uma abstração funcional $\lambda x.\, t$ tem tipo $\tau \to \rho$, em um dado contexto Γ, se é possível deduzir que t tem tipo ρ, em um contexto análogo a Γ, mas estendido de modo a associar a x o tipo τ. A regra (LET) determina que uma expressão $\texttt{let}\ x = u\ \texttt{in}\ t$ tem tipo ρ, em um contexto Γ, se podemos deduzir que u tem tipo σ nesse contexto, e que t tem tipo ρ em um contexto que consiste em Γ estendido com a atribuição de tipo σ para a variável x.

A regra (INST) especifica que o sistema é *predicativo*, isto é, variáveis de tipo apenas podem ser instanciadas para tipos monomórficos. Em um sistema *impredicativo*, tal como o sistema F por exemplo, variáveis de tipos podem ser instanciadas para tipos polimórficos. A regra (GEN) possibilita a generalização do tipo de uma expressão por meio de quantificação sobre variáveis de tipo que não ocorram livres no contexto.

A Figura 2.4 ilustra uma derivação de tipo no sistema HM (nesta derivação, usamos $\Gamma_y = \{y : \forall b.\, b \to b\}$).

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \beta \vdash x : b}\,\textsc{(VAR)}}{\vdash \lambda x.x : b \to b}\,\textsc{(ABS)}}{\vdash \lambda x.\, x : \forall b.\, b \to b}\,\textsc{(GEN)} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma_y \vdash y : \forall b.\, b \to b}\,\textsc{(VAR)}}{\Gamma_y \vdash y : [b \mapsto a \to a](b \to b)}\,\textsc{(INST)} \qquad \dfrac{\dfrac{}{\Gamma_y \vdash y : \forall b.\, b \to b}\,\textsc{(VAR)}}{\Gamma_y \vdash y : [b := a](b \to b)}\,\textsc{(INST)}}{\Gamma_y \vdash y\ y : a \to a}\,\textsc{(APP)}}{\vdash \texttt{let y = } \lambda x.\, x\ \texttt{in}\ y\ y : a \to a}\,\textsc{(LET)}$$

Figura 2.4. Árvore de derivação de tipo

A regra (ANNOT) não é incluída na maioria das apresentações do sistema HM, uma vez que anotações de tipo não são requeridas nesse sistema. Optamos entretanto por incluir anotações de tipo, com o objetivo de facilitar o entendimento do sistema de tipos proposto no capítulo 4, no qual anotações de tipo possuem papel fundamental.

Um termo $(t :: \sigma)$ é um termo cujo tipo é explicitamente anotado pelo programador. Por exemplo, considere o termo:

$$(\lambda x.\, x) :: \forall a.\, [a] \to [a]$$

Esse termo é bem tipado, uma vez que o tipo mais geral de $(\lambda x.\, x)$ é $\forall a.\, a \to a$, o qual é mais geral do que o tipo anotado $\forall a.\, [a] \to [a]$. A anotação é uma *restrição* de tipo, porque o termo cujo tipo é anotado apenas pode ser usado com o tipo especificado (ou com tipos que são instâncias deste). Por exemplo, a anotação invalida a aplicação $((\lambda x.\, x) :: \forall a.\, [a] \to [a])\ \texttt{True}$, embora $(\lambda x.\, x)$ possa ser aplicado a True.

No sistema HM, a regra (ANNOT) é bastante simples: ela apenas requer que o tipo anotado para um termo — $(t :: \sigma)$ — de fato possa ser derivado para esse termo.

2.2.2 Sistema de Tipos Dirigido por Sintaxe

Inferência de tipos pode ser definida como o problema de reconstruir anotações de tipo para um termo não explicitamente tipado, ou que inclui apenas anotações parciais de tipo. Em outras palavras, consiste no problema de determinar, dado um termo t e um contexto de tipos inicial Γ_0 (possivelmente vazio) se existe uma tipagem (Γ, σ) tal que $\Gamma \vdash e : \sigma$ e $\Gamma_0 \subseteq \Gamma$.

Esse problema é importante tanto do ponto de vista teórico como pragmático, já que desejamos programar em linguagens estaticamente tipadas, em que erros de tipo são detectados em tempo de compilação, mas desejamos incluir o mínimo de anotações de tipo em programas, já que isso é em geral tedioso e anotações de tipo excessivas podem prejudicar a clareza de programas.

No sistema apresentado na Figura 2.3, cada regra tem na conclusão uma forma sintática distinta, exceto as regras de generalização (GEN) e de instanciação (INST). Como essas regras têm a mesma forma sintática como premissa e como conclusão, elas podem ser aplicadas em qualquer instante da derivação; por exemplo, poderíamos alternar (GEN) e (INST) indefinidamente. Portanto é difícil obter um *algoritmo* de inferência de tipos a partir dessas regras, não sendo claro que regras usar, e em que ordem, para derivar um julgamento de tipo para um dado termo t em um contexto Γ.

Se todas as regras têm uma forma sintática distinta em sua conclusão, dizemos que o sistema está na forma *dirigida por sintaxe*, o que determina completamente a forma da árvore de derivação de tipo para um termo particular t. Isso é desejável, pois significa que o algoritmo de inferência de tipos pode ser dirigido pela sintaxe de termos, não sendo necessária uma busca exaustiva por uma derivação de tipo válida.

A Figura 2.5 apresenta uma forma alternativa de regras de derivação de tipos dirigida por sintaxe para o sistema HM. [2] Nessa formulação, tipos polimórficos ocorrem apenas no contexto de tipos. Os locais em que generalizações e instanciações são requeridas são agora determinados pela sintaxe de termos: a instanciação de tipos ocorre apenas na regra (VAR)e a generalização ocorre apenas na (LET). Portanto, as regras derivam julgamentos da forma $\Gamma \vdash t : \rho$, onde ρ é um tipo *monomórfico*.

[2] Uma prova da equivalência entre esse sistema e o sistema não dirigido por sintaxe apresentado anteriormente pode ser encontrada em [7].

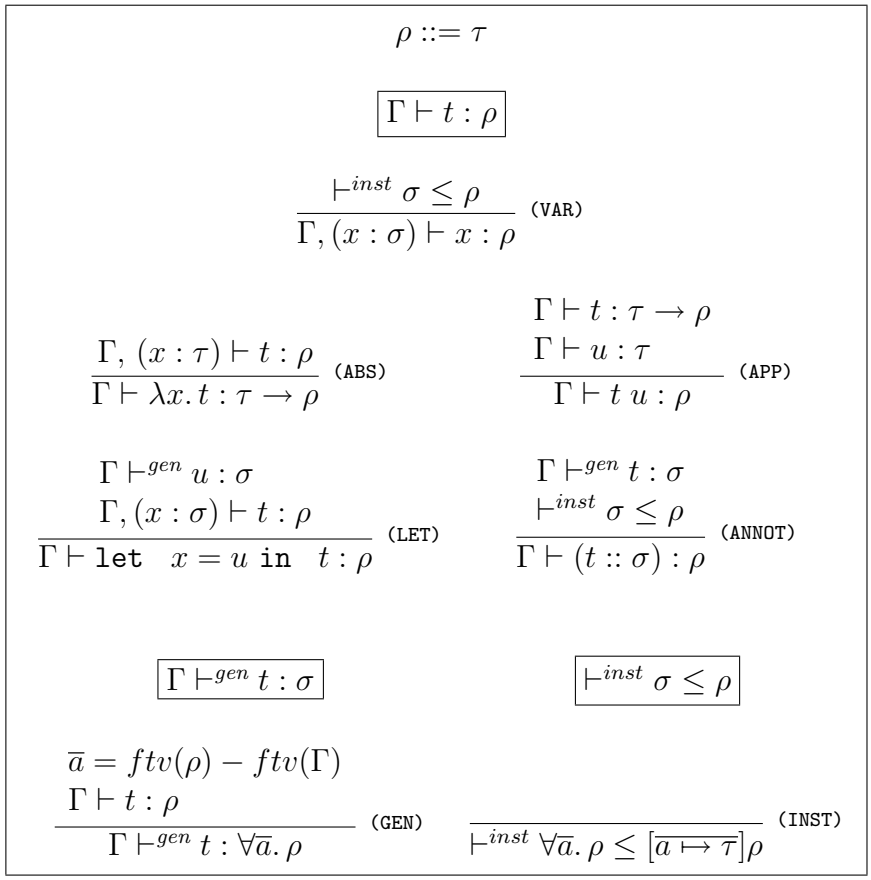

Figura 2.5. Sistema de tipos HM dirigido por sintaxe

A instanciação é tratada por meio do julgamento auxiliar

$$\vdash^{inst} \sigma \leq \rho$$

que significa que os quantificadores de σ podem ser instanciados de modo a obter ρ. Se $\vdash^{inst} \sigma \leq \rho$ dizemos que ρ é uma instância de σ (e que σ pode ser instanciado para o tipo ρ).

De forma dual, a generalização é tratada por meio do julgamento auxiliar

$$\Gamma \vdash^{gen} t : \sigma$$

que infere um tipo polimórfico para um termo, sendo este julgamento usado na regra (LET) para atribuir tipo à variável definida nessa construção. A regra (GEN) especifica que as variáveis de tipo quantificadas $\overline{a}$ devem ser exatamente as variáveis que ocorrem livres em ρ e não ocorrem livres em Γ: não há necessidade de generalizar sobre variáveis que não ocorram livres em ρ e, por outro lado, é necessário generalizar o máximo

possível.

2.2.3 Inferência de Tipos

As regras do sistema apresentado na Figura 2.5 não definem um *algoritmo*, já que em várias delas ocorre um tipo τ não especificado: por exemplo, o tipo τ que ocorre como tipo do parâmentro da lambda abstração, na regra `(ABS)`. Dado um contexto de tipos vazio, e o termo $\lambda x.\, x$, os seguintes julgamentos de tipo seriam válidos, escolhendo o tipo τ como sendo `Int`, `[`a`]` ou a, respectivamente:

$$\begin{aligned} &\vdash (\lambda x.\, x) : \texttt{Int} \rightarrow \texttt{Int} \\ &\vdash (\lambda x.\, x) : \texttt{[}a\texttt{]} \rightarrow \texttt{[}a\texttt{]} \\ &\vdash (\lambda x.\, x) : a \rightarrow a \end{aligned}$$

É claro que o algoritmo de inferência de tipos deve inferir o último desses tipos, pois é o tipo mais geral para $(\lambda x.\, x)$ (a menos de quantificação sobre a variável a). Do mesmo modo, na regra `(INST)` precisamos adivinhar os tipos $\overline{\tau}$ para os quais as variáveis quantificadas $\overline{a}$ devem ser instanciadas.

A idéia para obter o algoritmo de inferência de tipos é utilizar novas variáveis de tipo (*fresh*) em lugar de tipos não especificados, determinando o tipo requerido posteriormente, por meio de *unificação* [44]. A *unificação* de dois tipos (monomórficos) τ_1 e τ_2, escrita como $\mathit{unify}(\tau_1 = \tau_2)$, determina a substituição mais geral S que unifica esses dois tipos, caso tal substituição exista Mais formalmente, se $S = \mathit{unify}(\tau_1 = \tau_2)$, então S satisfaz as seguintes propriedades:

i) $S\tau_1 = S\tau_2$;

ii) para toda substituição S' tal que $S'\tau_1 = S'\tau_2$, existe uma substituição R tal que $S' = R \circ S$ (onde "$\circ$" denota composição funcional).

O algoritmo de inferência de tipos para `HM` foi proposto originalmente por Damas e Milner [10]. Esse algoritmo é apresentado na Figura 2.6, como um conjunto de regras de inferência para julgamentos da forma

$$\Gamma \vdash t : (\rho, \Gamma')$$

significando que a tipagem (Γ', ρ) é inferida para o termo t, dado um contexto de tipos inicial Γ.

Note que variáveis *fresh* são utilizadas nas regras `(ABS)` e `(VAR)`, em lugar dos tipos indeterminados τ Nesta última regra por meio da redefinição da relação de ins-

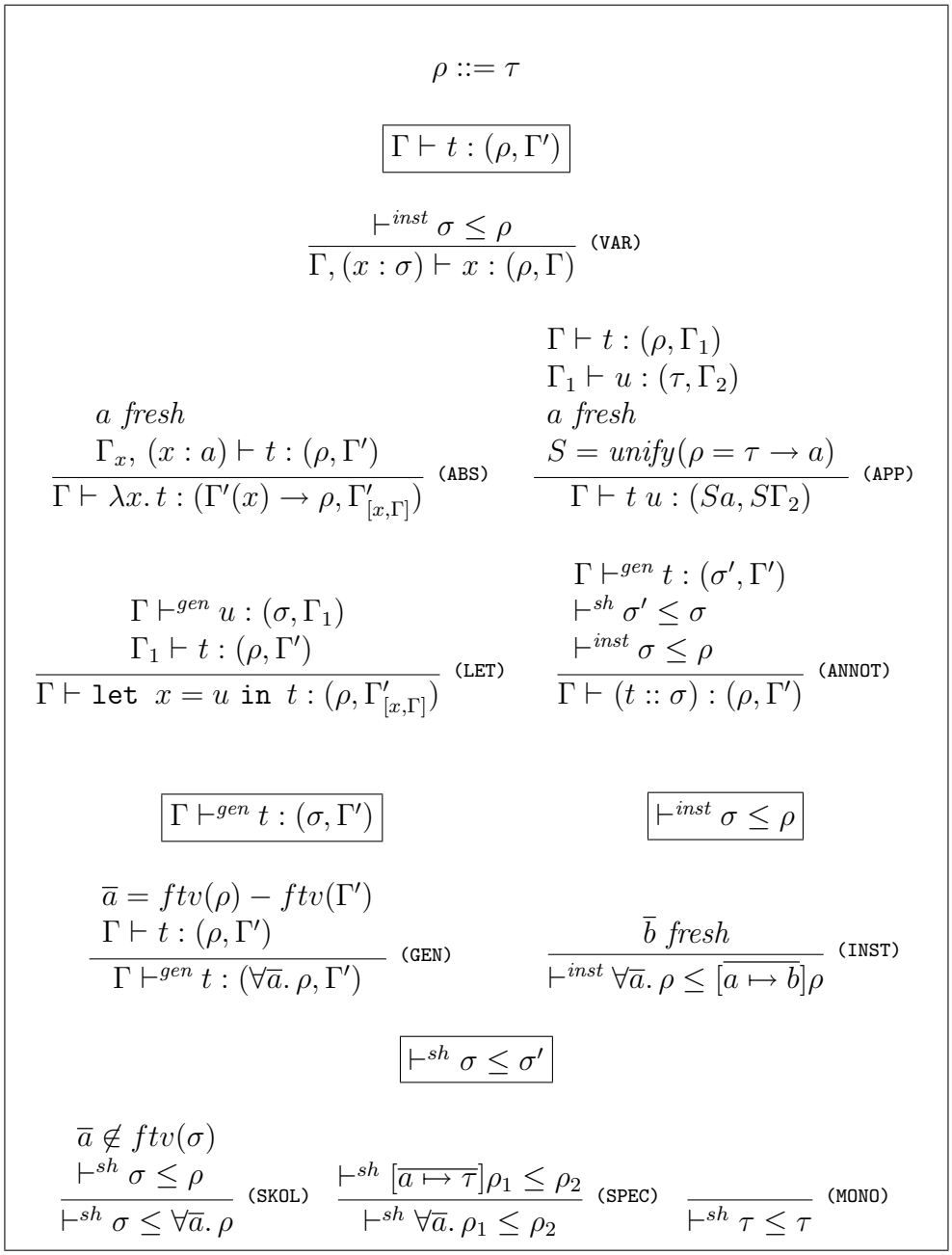

Figura 2.6. Inferência de tipos para HM

tanciação ($\vdash^{inst} \sigma \leq \tau$), na qual as variáveis quantificadas do tipo a ser instanciado são substituídas por variáveis *fresh*. A determinação do tipo requerido é feita por meio do processo de unificação, na regra (APP).

Na regra (ABS), o tipo da expressão que define o corpo da função é inferido no contexto de tipos $\Gamma_x, (x : a)$, significando que qualquer atribuição de tipo para uma variável de nome igual ao do parâmetro da função (x) é retirada do contexto, sendo

introduzido nesse contexto o tipo $(x : a)$, onde a é uma variável *fresh*. O tipo da variável x que eventualmente ocorra no escopo externo à lambda-abstração é recuperado no contexto resultante ($\Gamma'_{[x,\Gamma]}$), na conclusão dessa regra. O mesmo processo é usado na regra `(LET)`, para restringir o escopo da variável definida nessa construção.

No sistema de tipos apresentado na Figura 2.3, a regra para anotação de tipo é bastante simples: ela apenas requer que um termo $(t :: \sigma)$ de fato tenha tipo σ. Na inferência de tipos, a regra `(ANNOT)` verifica o tipo anotado em três passos:

- Obter o *tipo mais geral* para t, ou seja σ', usando $\vdash^{gen}$;
- Esse tipo pode ser diferente do tipo anotado σ, pois esse último não necessariamente é o tipo mais geral de T. Portanto, o próximo passo é verificar se o tipo σ' é mais geral (ou mais polimórfico) do que σ, o que é feito por meio do julgamento $\vdash^{sh} \sigma' \leq \sigma$ (o superescrito sh indica *subsunção*);
- Finalmente, instanciar σ, usando $\vdash^{inst}$.

Diferentemente de $\vdash^{inst}$, o julgamento $\vdash^{sh} \sigma' \leq \sigma$ compara tipos polimórficos. Por exemplo:

$$\begin{aligned} \texttt{Int} \to \texttt{Bool} &\leq \texttt{Int} \to \texttt{Bool} \\ \forall a.\, a \to a &\leq \texttt{Int} \to \texttt{Int} \\ \forall a.\, a \to a &\leq \forall b.\, \texttt{[}b\texttt{]} \to \texttt{[}b\texttt{]} \\ \forall ab.\, (a, b) \to (b, a) &\leq \forall c.\, (c, c) \to (c, c) \end{aligned}$$

Na definição do julgamento $\vdash^{sh} \sigma' \leq \sigma$, a regra `(MONO)` lida com o caso trivial de dois tipos monomórficos. No caso mais geral, provar $\vdash^{sh} \sigma' \leq \sigma$, implica mostrar que toda instância de σ é também uma instância de σ'. Formalmente, para provar $\vdash^{sh} \forall \overline{a}.\, \rho' \leq \forall \overline{b}.\, \rho$ devemos provar que:

$$\forall \overline{\tau_b}.\, \exists \overline{\tau_a}.\, [\overline{a \mapsto \tau_a}]\rho' \leq [\overline{b \mapsto \tau_b}]\rho$$

Para isso, a regra `(SPEC)` nos permite instanciar as variáveis de tipo quantificadas de σ', de maneira a obter ρ. Mas como podemos verificar que σ' pode ser instanciado, por meio de `(SPEC)`, para *qualquer* instância de σ? A idéia é instanciar as variáveis quantificadas de σ para novas constantes de tipo arbitrárias (*fresh*), denominadas *constantes de Skolem*, e então verificar se σ' pode ser instanciado para o tipo ρ obtido por esse processo de *skolemização*. De fato, a regra `(SKOL)` não instancia as variáveis de σ para novas constantes de tipo arbitrárias, mas, equivalentemente, apenas verifica que essas variáveis são novas em relação a σ' (isto é, que não ocorrem no conjunto de variáveis livres de σ', considerando-se possível renomeação de variáveis ligadas em σ').

Note que a regra (SKOL) deve ser aplicada antes da regra (SPEC), uma vez que esta última pressupõe um tipo ρ do lado direito de $\leq$. Ou seja, primeiro instanciamos σ para constantes de Skolem, e então escolhemos como instanciar σ' de modo a casar com o tipo obtido por skolemização de σ. Por exemplo, para provar que

$$\forall a.\, a \to a \leq \forall b\, c.\, (b, c) \to (b, c)$$

primeiro usamos a regra (SKOL) para skolemizar b e c, verificando que b e c não ocorrem livres em $\forall a.\, a \to a$, e então usamos a regra (SPEC) para instanciar a para (b, c). A derivação tem a seguinte forma:

$$\dfrac{\dfrac{\dfrac{}{[a \mapsto (b,c)]a \to a \leq (b,c) \to (b,c)}\,\text{(MONO)}}{\forall a.\, a \to a \leq (b,c) \to (b,c)}\,\text{(INST)}}{\forall a.\, a \to a \leq \forall b\, c.\, (b,c) \to (b,c)}\,\text{(SKOL)}$$

A relação $\vdash^{sh} \sigma' \leq \sigma$ pode ser implementada usando skolemização e unificação, como se mostra a seguir, onde as variáveis $\overline{c_K}$ representam novas constantes de tipo, que portanto não podem ocorrer no domínio da substituição retornada pela unificação.

$$\dfrac{\begin{array}{l}\vdash^{inst} \sigma' \leq \rho' \\ \overline{c_K}\ \mathit{fresh} \\ S = \mathit{unify}(\rho' = [\overline{a \mapsto c_K}]\rho)\end{array}}{\vdash^{sh} \sigma' \leq \forall \overline{a}.\, \rho}\,\text{(SKOL)}$$

Sistemas de tipos polimórficos possibilitam, em geral, obter diferentes derivações de tipo para uma dada expressão em um determinado contexto. É importante portanto que o tipo inferido para uma expressão, em um dado contexto, seja o tipo mais geral que pode ser derivado para a expressão nesse contexto, já que a semântica de expressões deve poder ser dada em termos da derivação do tipo inferido. Para formalizar o enunciado dessa propriedade de tipo para o algoritmo acima, é necessário introduzir uma definição de ordem entre contextos de tipo, a qual é definida em termos da relação de ordem entre tipos $\vdash^{sh}$:

$$\dfrac{\mathit{dom}(\Gamma) = \mathit{dom}(\Gamma') \text{ e para todo } (x : \sigma) \in \Gamma \text{ e } (x : \sigma') \in \Gamma',\ \vdash^{sh} \sigma \leq \sigma'}{\vdash^{sh} \Gamma \leq \Gamma'}$$

A inferência de tipos no sistema HM satisfaz as seguintes propriedades importantes, que garantem a sua coerência em relação ao sistema de tipos (para uma prova dessas

propriedades veja, por exemplo [40, 35]):

Teorema 2.2.1 (Correção) *Para todo termo t, se $\Gamma \vdash t : (\rho, \Gamma')$ então $\Gamma' \vdash t : \rho$*

Teorema 2.2.2 (Tipo Principal) *Para todo termos t, se $\Gamma \vdash t : \sigma$, então $\Gamma \vdash t : (\tau, \Gamma')$, onde $\vdash^{sh} \Gamma' \leq \Gamma$ e $\Gamma' \vdash^{gen} t : (\sigma', \Gamma')$ $\vdash^{sh} \sigma' \leq \sigma$.*

2.2.4 Semântica

De maneira geral, existem dois estilos de atribuir significado a termos de uma linguagem tipada, usualmente denomidados *intrínseco* ou *extrínseco* ou, alternativamente, *estilo Church* e *estilo Curry*[2], respectivamente. No estilo Church, a semântica é definida sobre derivações no sistema, sendo portanto atribuído significado apenas a termos bem tipados. Nessa visão, a noção de tipo é anterior à noção de significado e, portanto, não faz sentido a questão "qual é o comportamento de um termo não tipado?".

Uma semântica em estilo *Curry*, por outro lado, atribui significado aos termos independentemente de seu tipo (portanto, termos não envolvem anotações de tipo), sendo esse significado o mesmo que na versão não tipada da linguagem. Em outras palavras, nessa visão primeiro são definidos os termos (não tipados) e sua semântica (por meio de regras de redução), sendo posteriormente definido um sistema de tipos, que rejeita termos cujo comportamento é indesejado.

A semântica de expressões do sistema `HM` é definida na Figura 2.7, em estilo *Church*, sobre derivações do sistema de tipo dirigido por sintaxe:

$$\Gamma \vdash t : \rho \rightsquigarrow t'$$

onde "o termo t tem tipo ρ no contexto Γ, tendo signicado t'. Nessa Figura, omitimos as definições das relações $\vdash^{inst}$ e $\vdash^{gen}$, uma vez que não são relevantes para a semântica de termos.

O mecanismo fundamental para a definição de semântica dos termos de `HM` é a β-redução[1], que provê uma noção de um passo de avaliação (ou computação) dos termos da linguagem. Uma aplicação da forma $(\lambda x.t)\, u$ *reduz* para a expressão $[x \mapsto u]t$, isto é, para o termo obtido pela substituição no termo t de todas as ocorrências *livres* da variável x pelo termo u, sendo essa substituição válida apenas se não resulta em captura de variáveis livres em u. Por Exemplo a substituição $[x \mapsto y](\lambda y.xy)$ não seria válida, já que resultaria no termo $(\lambda y.yy)$, em que a variável livre x é capturada nesse processo de substituição.[3]

[3] Veja, por Exemplo [1, 2, 40] para uma definição precisa das noções de variáveis livres e substituição sem captura de variáveis.

$$\rho ::= \tau$$

$$\boxed{\Gamma \vdash t : \rho \rightsquigarrow t'}$$

$$\frac{\vdash^{inst} \sigma \leq \rho}{\Gamma, (x : \sigma) \vdash x : \rho \rightsquigarrow x}\ \texttt{(VAR)}$$

$$\frac{\Gamma, (x : \tau) \vdash t : \rho \rightsquigarrow t'}{\Gamma \vdash \lambda x.\, t : \tau \to \rho \rightsquigarrow \lambda x.\, t'}\ \texttt{(ABS)} \qquad \frac{\Gamma \vdash t : \tau \to \rho \rightsquigarrow (\lambda x.\, t') \quad \Gamma \vdash u : \tau \rightsquigarrow u'}{\Gamma \vdash t\ u : \rho \rightsquigarrow [x \mapsto u']t'}\ \texttt{(APP)}$$

$$\frac{\Gamma \vdash^{gen} u : \sigma \rightsquigarrow u' \quad \Gamma, (x : \sigma) \vdash t : \rho \rightsquigarrow t'}{\Gamma \vdash \texttt{let}\ \ x = u\ \texttt{in}\ \ t : \rho \rightsquigarrow [x \mapsto u']t'}\ \texttt{(LET)} \qquad \frac{\Gamma \vdash^{gen} t : \sigma' \rightsquigarrow t' \quad \vdash^{sh} \sigma' \leq \sigma \quad \vdash^{inst} \sigma \leq \rho}{\Gamma \vdash^{gen} (t :: \sigma) : \rho \rightsquigarrow t'}\ \texttt{(ANNOT)}$$

Figura 2.7. Semântica do sistema `HM`

A propriedade mais fundamental de um sistema de tipos é *correção* (em inglês, *safety* ou *soundness*), que garante que a avaliação de um termo bem tipado não resulta em "erro de tipo": sua execução não falha inesperadamente. Isso é determinado estaticamente, isto é, sem que o programa seja executado, o que provê ao programador uma garantia parcial de robustez ou segurança do programa.

A prova de que o sistema `HM` é *correto* em relação à semântica definida na Figura 2.7 deriva diretamente das duas seguintes propriedades mais fundamentais (para uma prova dessas propriedades, veja, por Exemplo, [35, 40]).

Teorema 2.2.3 (*Progress*) *: Seja t um termo fechado da linguagem tal que $\Gamma \vdash t : \rho$. Então $\Gamma \vdash t : \rho \rightsquigarrow t'$, para algum termo t'.*

Intuitivamente, essa propriedade garante que todo termo bem tipado t pode ser avaliado segundo as regras de avaliação de expressões da linguagem, ou seja, t é um valor ou pode ser reduzido para algum termo t'.

Teorema 2.2.4 (*Preservation*) *: Se $\Gamma \vdash t : \rho \rightsquigarrow t'$ então $\Gamma \vdash t' : \rho$.*

Intuitivamente, o esse teorema garante que o termo resultante da aplicação de um passo de avaliação sobre um termo bem tipado é também um termo bem tipado, e tem o mesmo tipo principal que o tipo original.

2.3 Polimorfismo de Rank Superior

Sistemas com suporte para polimorfismo de rank arbitrário (ou rank-n) são mais expressivos que o sistema `HM`, possibilitando definir funções com parâmetros de tipo polimórfico, tais como no Exemplo 1.1 (3), apresentado anteriormente. Nesse Exemplo, o parâmetro `g` é usado polimorficamente no corpo da função `foo`, sendo aplicado a listas de booleanos e listas de caracteres, o que é refletido na anotação de tipo provida para a função `foo`. A aplicação `foo reverse`, por Exemplo, seria válida, produzindo como resultado `([False, True], ['c','a','b'])`. A definição da função `foo`, entretanto, não seria bem tipada no sistema `HM`, pois esse sistema requer que parâmetros de funções tenham tipo *monomórfico* — o sistema de tipos pode atribuir a `g` o tipo `[Int]` $\rightarrow$ `[Int]` ou `[Bool]` $\rightarrow$ `[Bool]`, mas não o tipo polimórfico $\forall a.$ `[a]` $\rightarrow$ `[a]`, o que apenas seria permitido em um sistema com suporte para polimorfismo de rank superior.

Aplicações mais interessantes de tipos polimórficos de rank superior são descritas em [45, 38, 50]. Coloca-se portanto a questão: "é possível estender o sistema `HM` de modo a considerar tipos polimórficos de rank superior, mas sem que o sistema resultante, ou o mecanismo de inferência de tipos, seja demasiadamente complicado?" Alguns trabalhos que buscam dar resposta a essa questão são discutidos na Seção 2.3.2.

A referência fundamental para sistemas de polimorfismo de rank arbitrário (ou rank-n) é o sistema `F` proposto simultaneamente por Girard[14, 18] e Reynolds[43], também chamado de lambda calculus de segunda ordem, devido a sua correspondência com a Lógica Intuicionista de Segunda Ordem (veja [14, 18]). O sistema `F` não é entretanto adequado como base para a implementação de suporte a polimorfismo de rank superior em linguagens de programação, porque não possui a propriedade de *tipo principal* e a inferência de tipos nesse sistema é indecidível [55].

O sistema `F` é comumente usado como linguagem objeto para a definição de semântica de extensões do sistema `HM`, tanto para suporte a polimorfismo de rank superior como para polimorfismo de sobrecarga, e tem sido usado como linguagem objeto em compiladores de linguagens funcionais modernas.

2.3.1 Sistema `F`

A linguagem de tipos (σ) do sistema `F` é como definida anteriormente (na Seção 2.1): note que $\rho = \tau \mid \sigma \rightarrow \sigma$. A linguagem de termos e expressões do sistema `F` (explicitamente tipado) é apresentada na Figura 2.8. Nessa linguagem, cada ocorrência ligada de variável é anotada com seu respectivo tipo. Uma *aplicação de tipo explícita* ($e\,\sigma$) especifica os tipos que instanciam uma função polimórfica f e uma *abstração de tipo*

Expressões	e, f	$::=$	x	Variáveis
		$\mid$	$\Lambda a.\, e$	Abstração de tipo
		$\mid$	$\lambda(x : \sigma).e$	Abstração funcional
		$\mid$	$(e\ \sigma)$	Aplicação de tipo
		$\mid$	$(f\ e)$	Aplicação de valor
		$\mid$	$\texttt{let}\ (x : \sigma) = e_1\ \texttt{in}\ \ e_2$	Definição local

Figura 2.8. Sintaxe da linguagem do sistema `F`

$$\boxed{\rho = \tau \mid \sigma \to \sigma}$$

$$\boxed{\Gamma \vdash e : \sigma}$$

$$\frac{}{\Gamma, (x : \sigma) \vdash x : \sigma}\ \texttt{(VAR)}$$

$$\frac{\Gamma, (x : \sigma') \vdash e : \sigma}{\Gamma \vdash \lambda(x :: \sigma').\, e : \sigma' \to \sigma}\ \texttt{(ABS)} \qquad \frac{\Gamma \vdash f : \sigma' \to \sigma \quad \Gamma \vdash e : \sigma'}{\Gamma \vdash (f\ e) : \sigma}\ \texttt{(APP)}$$

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \Lambda a.\, e : \forall a.\, \sigma}\ \texttt{(TABS)} \qquad \frac{\Gamma \vdash \Lambda a.\, e : \forall a.\, \sigma}{\Gamma \vdash (e\ \sigma') : [a \mapsto \sigma']\sigma}\ \texttt{(APP2)}$$

$$\frac{\Gamma \vdash e_1 : \sigma' \qquad \Gamma, (x : \sigma') \vdash e : \sigma}{\Gamma \vdash \texttt{let}\ (x :: \sigma') = e_1 \texttt{in}\ e_2 : \sigma}\ \texttt{(LET)}$$

Figura 2.9. Sistema `F`

explícita $(\Lambda a.\, e)$ especifica como e onde é feita generalização sobre variáveis de tipo. Definições locais são incluídas na linguagem apenas por conveniência, embora não sejam necessárias, já que uma definição local $\texttt{let}\ (x :: \sigma) = e_1\ \texttt{in}\ e_2$ é equivalente à aplicação $((\lambda(x :: \sigma).\, e_2)\ e_1)$.

As regras de derivação de tipo para expressões do sistema `F` são apresentadas na Figura 2.9.

No sistema `F` é possível atribuir tipo, por Exemplo, à expressão $\lambda x.x\ x$ (de fato, uma quantidade infinitamente enumerável de tipos), tal como, por Exemplo, os tipos $(\forall a.\, a) \to (\forall a.\, a)$ ou $(\forall a.\, a \to a) \to (\forall a.\, a \to a)$. Note que nenhum desses dois tipos é instância um do outro; e é possível mostrar que nenhum outro tipo derivável para essa expressão é mais geral do que ambos (isto é, pode ser instanciado para cada um deles), o que significa que o sistema `F` não possui a propriedade de tipo principal.

2.3.2 Estendendo HM com polimorfismo de rank-n

Alguns trabalhos recentes propõem extensões ao sistema HM no sentido de prover polimorfismo de rank superior, buscando manter compatibilidade com HM na ausência de anotações de tipo (isto é, nesse caso, o tipo inferido deve ser o mesmo que o obtido em HM), mas usar anotações de tipo explícitas para possibilitar a definição de funções de ordem superior, buscando prover a mesma expressividade do sistema F. É claro que é também desejável que os requerimentos de anotações de tipo em programas sejam mínimos. Para isso, a idéia é propagar a informação provida pelas anotações de tipo ao longo do programa (durante a inferência de tipos), evitando assim o requerimento de anotações de tipo redundantes.

A idéia de propagar anotações de tipo em programas, usando essa informação para guiar a inferência de tipos foi primeiramente proposta por Pierce and Turner [41], sendo chamada de *inferência de tipos local*. O termo *inferência de tipo bidirecional* é usado em [38], trabalho que propõe um algoritmo para inferência de tipos no sistema de polimorfismo de rank superior proposto por Odersky e Laufer [37], um dos primeiros trabalhos no sentido de estender HM com polimorfismo de rank superior.

O sistema de tipos proposto por Odersky e Laufer, diferentemente do sistema F, admite instanciação de quantificadores que ocorrem no lado esquerdo do construtor funcional (*deep instantiation*), levando em conta, naturalmente, a contra-variância deste construtor. O sistema proposto neste trabalho é *predicativo* (variáveis quantificadas podem ser instanciadas apenas para tipos monomórficos), o que restringe bastante a sua expressividade. Considere, por Exemplo, a aplicação `length ids`, onde os tipos de `length` e `ids` são:

$$\texttt{length} :: \quad \forall a.\, [a] \rightarrow \texttt{Int}$$
$$\texttt{ids} :: \quad [\forall c.\, c \rightarrow c]$$

Para que a aplicação a aplicação `length ids` seja bem tipada, a variável quantificada (a) no tipo de `length` deve poder ser instanciada para o tipo polimórfico $\forall c.\, c \rightarrow c$. Outra desvantagem do sistema de Odersky e Laufer é requerer anotações de tipo redundantes e excessivas, problema mais tarde contornado em [38], por meio do mecanismo de propagação de tipos, ou *inferência de tipos bidirecional*.

O sistema $F_{\leq^n}$, proposto por Mitchell [36], permite *deep instantiation* e é *impredicativo* (variáveis quantificadas podem ser instanciadas para tipos polimórficos), e atribui tipo a toda expressão bem tipada no sistema F. A vantagem de $F_{\leq^n}$ em relação a F é que o primeiro possui a propriedade de tipo principal, o que leva os proponentes deste sistema a argumentar que ele seria, portanto, mais adequado como base para a

implementação de polimorfismo de rank superior em linguagens de programação.

A inferência de tipos para $F_{\leq^n}$, mesmo na presença de anotações de tipo, é entretanto extremamente complicada. O algoritmo de inferência de tipos apresentado em [11] é baseado em $F_{\leq^n}$, mas entretanto resolve o problema de decidir adequadamente, em todos os casos, quando variáveis quantificadas devem ser instanciadas predicativamente ou impredicativamente.

Para entender esse problema, considere, por Exemplo a aplicação `choose id`, onde os tipos de `choose` e `id` são:

$$\begin{aligned} &\texttt{choose} :: \quad \forall a.\, a \to a \to a \\ &\texttt{id} :: \quad \forall b.\, b \to b \end{aligned}$$

No sistema `HM`, o tipo inferido para a expressão `(choose id)` seria $\forall b.\, (b \to b) \to (b \to b)$. Entretanto, se o tipo de `choose` pode ser instanciado impredicativamente, poderíamos também tipar essa expressão com o tipo $(\forall b.\, b \to b) \to (\forall b.\, b \to b)$, e nenhum desses tipos é principal (esses dois tipos são incomparáveis no sistema `F`).

No sistema `MLF`, apresentado em [28, 29], o problema de "adivinhar" quando deve ser feita instanciação predicativa ou impredicativa é resolvido estendendo-se a linguagem de tipos do sistema `F`, de modo a incluir tipos com *restrições de instanciação*. Por Exemplo, nessa linguagem, poderia ser atribuído à expressão `choose id` o tipo principal $\forall(a \geq \forall b.\, b \to b).\, a \to a$. Novamente, a inferência de tipos é extremamente complicada, o que motivou a simplificação de `MLF` proposta em [31, 32], na qual é requerida anotação explícita de tipo para todo parâmetro de função com tipo polimórfico, sendo essa informação usada para simplificar a inferência de tipos. Esse último trabalho considera também a possibilidade de restringir o sistema, de modo que anotações de tipo sejam restritas a tipos do sistema `F` (e não tipos de `MLF`).

Os trabalhos apresentados em [51, 52] também buscam contornar esse problema, mas procurando manter a mesma linguagem de tipos do sistema `F`. No sistema propoto nesses trabalhos — `FPH` — o tipo inferido para a aplicação `choose id`, por Exemplo, seria o tipo principal dessa expressão no sistema `HM`, ou seja, $\forall a.\, (a \to a) \to (a \to a)$. O tipo $(\forall a.\, a \to a) \to (\forall a.\, a \to a)$ poderia ser inferido apenas no caso de ser provida a anotação explícita de tipo `(choose id)` $:: (\forall a.\, a \to a) \to (\forall a.\, a \to a)$.

2.4 Conclusão

Neste Capítulo fizemos uma revisão sobre sistemas de tipo com suporte a polimorfismo paramétrico, apresentando detalhadamente o sistema de Hindley-Milner, que provê su-

porte a polimorfismo de rank-1, e também discutimos algumas propostas de extensão a esse sistema no sentido de prover polimorfismo de rank-n. Como vimos, tais extensões possibilitariam a definição de uma função tal como a função `foo`, apresentada no Exemplo 1.1 (3), mediante anotação explícita do tipo do parâmetro dessa função. Considerando o tipo anotado para `foo` nesse Exemplo, seria válida, por Exemplo, a aplicação `foo reverse`.

O Capítulo 3 apresenta uma extensão do sistema de Hindley Milner para prover suporte a definições sobrecarregadas, a qual é baseada no mecanismo de classes de tipos de Haskell. Veremos que essas duas extensões não estão ainda adequadamente integradas: embora sistemas com polimorfismo de rank-n possibilitem definir uma função com parâmetro de tipo polimórfico, não é possível aplicar tal função a um argumento de tipo polimórfico restrito, ou seja, a uma expressão cuja definição envolve o uso de algum símbolo sobrecarregado.

Capítulo 3

Polimorfismo Restrito: Sobrecarga

O polimorfismo *ad hoc*, ou polimorfismo de *sobrecarga*, possibilita prover diferentes definições para um mesmo símbolo (ou variável), possivelmente com tipos distintos, sendo a definição com o tipo apropriado escolhida de acordo com o contexto no qual o símbolo é usado. Em outras palavras, a definição apropriada de uma função é escolhida de acordo com o tipo e a aridade dos argumentos ao qual a função é aplicada ou de acordo com o tipo requerido para o resultado dessa aplicação.[1]

Note que o polimorfismo de sobrecarga difere do polimorfismo paramétrico discutido anteriormente, no qual uma função polimórfica tem uma definição única, que opera de maneira uniforme independentemente do tipo da expressão na qual ela é usada. A combinação, em uma linguagem de programação, dessas duas formas de polimorfismo — paramétrico e de sobrecarga — resulta em grande expressividade da linguagem, como veremos em Exemplos apresentados neste capítulo.

Este capítulo apresenta uma extensão do sistema `HM` para suporte a sobrecarga, baseada no sistemas de classes de tipos de Haskell, que denominamos sistema `CTH`. Esse sistema é uma versão do sistema de tipos qualificados proposto por Mark Jones [19, 15], mas inclui também idéias propostas em [3, 4]. O sistema `CTH` constitui a base para a definição do sistema de tipos proposto no capítulo 4.

A Seção 3.1 provê uma introdução ao sistema de classes de tipos de Haskell. A Seção 3.2 apresenta a sintaxe de tipos e expressões do sistema [3], dorvante denominado `CTH` e a Seção 3.3 apresenta as regras de derivação de tipos nesse sistema. A inferência de tipos é apresentada na Seção 3.4. A semântica de expressões desse sistema é descrita

[1] Algumas linguagens de programação, tais como Java, possibilitam apenas a sobrecarga de símbolos que representam funções, de maneira que a definição apropriada possa ser selecionada de acordo com o tipo e número de argumentos ao qual a função é aplicada. Essa estratégia é denominada sobrecarga independente de contexto, enquanto que a estratégia adotada em Haskell, mais flexível, é denominada dependente de contexto[54].

na Seção 3.4.1.

3.1 Classes de tipos em Haskell

Considere a definição, em Haskell, de uma função `elem` que determina se um dado valor é elemento de uma lista dada:

```
elem ::  Eq a => a -> [a] -> Bool
elem y []     = False
elem y (x:xs) = y==x || elem y xs
```

Note que essa função se comporta de maneira uniforme para listas de elementos de qualquer tipo, exceto que é definida em termos do teste de igualdade `(==)`, que opera diferentemente para valores de tipos diferentes — em outras palavras, `(==)` é um símbolo sobrecarregado. A restrição de tipo `Eq a` que aparece no tipo dessa função indica essa dependência, e restringe a aplicação dessa função a listas de elementos para os quais a operação `(==)` é definida.

Em Haskell, a definição de um símbolo sobrecarregado é feita por meio do mecanismo classes de tipos. Esse mecanismo é ilustrado por meio do Exemplo a seguir, que apresenta a definição da classe `Eq`, que define os tipos dos símbolos `(==)` e `(/=)`:

```
class Eq a where
     (==) ::  a -> a -> Bool
     (/=) ::  a -> a -> Bool
```

A definição do operador `(==)` para valores de tipos específicos é provida por meio da definição de *instâncias* dessa classe. Para tipos básicos, tais como `Int`, a definição seria dada em uma definição de instância tal como a seguir, onde `primEqInt` é o teste de igualdade pré-definido para valores inteiros:

```
instance Eq Int where
     (==)     = primEqInt
     (a /= b) = not (a == b)
```

A declaração de instância a seguir exemplifica a definição de uma instância dessa classe para listas (polimórficas):

```
instance Eq a => Eq [a] where
    [ ]    == [ ]    = True
    (x:xs) == (y:ys) = (x == y) && (xs == ys)
```

Note que, nessa definição, o teste de igualdade para elementos de um tipo arbitrário, mas fixo, é usado para definir o teste de igualdade para listas de elementos desse tipo. Em face dessas definições, a operação `(==)` pode ser usada para comparar listas de inúmeros tipos, como por Exemplo `[Int]`, `[[Int]]`, e assim por diante.

Uma declaração de classe pode possivelmente incluir restrições de classe, como no Exemplo a seguir, estabelecendo desse modo uma *hierarquia* entre classes:

```
class Eq a => Ord a where
     (>) ::  a -> a -> Bool
     (>=) ::  a -> a -> Bool
```

A restrição de classe `Eq a` na definição da classe `Ord a` significa que apenas podem ser definidos como instâncias dessa classe tipos que sejam instância da classe `Eq a`.

O sistema de classes de tipos de Haskell [15] é originalmente baseado nos trabalhos propostos em [53, 21]. Outros trabalhos propõem extensões a esses primeiros, como a definição de classes de tipos com múltiplos parâmetros [25] ou de classes parametrizadas por construtores de tipos [22, 25], a definição de dependências funcionais [20, 24, 12, 49] ou alternativas para essa proposta [6, 4], ou analisam questões referentes a esses sistemas, tais como a inferência de tipos [3, 13, 47, 48], o problema de coerência ou ambiguidade [20, 4], ou abordagens de "mundo aberto" ou "mundo fechado" para a definição de instâncias de classes [5].

As seções a seguir apresentam um sistema de tipos para suporte a sobrecarga, juntamente com a semântica desse sistema e o algoritmo de inferência de tipos correspondente. O sistema de tipos aqui apresentado, referenciado como sistema `CTH`, uma versão baseada nos sistemas apresentados em [21] e [4].

3.2 Classes, Instâncias e Polimorfismo Restrito

A Figura 3.1 apresenta a sintaxe livre de contexto de tipos e expressões da linguagem do sistema `CTH`. Por simplicidade, nesse sistema não consideramos classes parametrizadas por construtores de tipo (e o problema relacionado de inferência de *kind* [22]), mas supomos que classes podem ser parametrizadas por múltiplas variáveis de tipo.

Um programa consiste de *declarações de classe* e *declarações de instância*, com

Tipos monomóficos	τ	::=	$a \mid \tau_1 \rightarrow \tau_2$
Tipos com restrições	ρ	::=	$P \Rightarrow \tau$
Tipos polimórficos	σ	::=	$\forall \overline{a}.\, \rho$
Predicado de classes	κ	::=	$C\,\overline{\tau}$
Contexto de predicados	P, Q	::=	$\emptyset \mid P, \kappa$
Contexto de tipos	Γ	::=	$\emptyset \mid \Gamma, (x : \sigma)$
Contexto de classes	$\Gamma^{\mathtt{c}}$	::=	$\emptyset \mid \Gamma^{\mathtt{c}}, (\mathit{class}\ P \Rightarrow \kappa) \mid \Gamma^{\mathtt{c}}, (\mathit{inst}\ P \Rightarrow \kappa)$
Programas	p	::=	$\overline{cD};\ \overline{iD};\ t$
Declaração de classe	cD	::=	$\mathtt{class}\ P \Rightarrow C\,\overline{\tau}\ \mathtt{where}\ \overline{x :: \tau}$
Declaração de instância	iD	::=	$\mathtt{instance}\ P \Rightarrow C\,\overline{\tau}\ \mathtt{where}\ \overline{x = e}$

Figura 3.1. Sintaxe da linguagem do sistema `CTH`

escopo global, e de uma expressão, sendo a sintaxe de expressões como no sistema `HM`.

Uma declaração de classe

$$\mathtt{class}\ P \Rightarrow C\,\overline{a}\ \mathtt{where}\ \{x_1 :: \tau'_1; \ldots; x_n :: \tau'_n\}$$

especifica o nome (C) e os parâmetros ($\overline{a}$) da classe, bem como as assinaturas dos símbolos sobrecarregados definidos nessa classe ($\{x_1 :: \tau'_1, \ldots, x_n :: \tau'_n\}$), podendo incluir restrições de classe (P). O tipo especificado para cada símbolo declarado na classe deve incluir todas as variáveis que são parâmetros da classe. [2]

Uma declaração de instância

$$\mathtt{instance}\ P \Rightarrow C\,\overline{\tau}\ \mathtt{where}\ \{x_1 = e_1; \ldots; x_n = e_n\}$$

especifica o nome da classe para a qual a instância é definida (C), assim como os tipos para os quais são instanciados os parâmetros da classe ($\overline{\tau}$), e provê uma definição para cada um dos símbolos sobrecarregados da classe. É claro que essa declaração de instância apenas é válida se existir a declaração correspondente da classe C e i) o cabeçalho da declaração de instância ($C\,\overline{\tau}$) corresponde a uma instanciação do cabeçalho da classe C (obtida por uma substituição dos parâmetros da classe por tipos apropriados); ii) pode-se verificar que a definição provida para cada símbolo da classe tem o tipo apropriado (ou seja, o tipo obtido instanciando-se o tipo provido na assinatura

[2] Algumas restrições devem ser impostas às definições de classe, tal como não circularidade da hieraquia de classes, de maneira a garantir decidibilidade da relação de provabilidade de predicados de classe (veja [47, 48]).

$$\frac{P \models Q \quad (\mathit{class}\ Q \Rightarrow \kappa) \in \Gamma^{\mathsf{c}} \quad \kappa' \in Q}{P \models \{\kappa'\}}\ \mathit{super}$$

$$\frac{P \models Q \quad (\mathit{inst}\ Q \Rightarrow \kappa) \in \Gamma^{\mathsf{c}}}{P \models \{\kappa\}}\ \mathit{inst}$$

Figura 3.2. Provabilidade de predicados de classe

desse símbolo conforme definido na declaração de instância).

Um tipo σ inclui agora um conjunto (possivelmente vazio) de *predicados de classe*, da forma $C\,\overline{\tau}$, que impõem restrições sobre os tipos para os quais σ pode ser instanciado. A meta variável κ é usada para denotar um predicado de classe; P e Q denotam conjuntos de predicados de classe; escrevemos P, κ para denotar o conjunto $P \cup \{\kappa\}$ e P, Q para denotar $P \cup Q$. Um tipo $\forall \overline{a}.\, P \Rightarrow \tau$ representa o conjunto dos tipos $[\overline{a \mapsto \tau'}]P \Rightarrow [\overline{a \mapsto \tau'}]\tau$ tais que os predicados de classe $[\overline{a \mapsto \tau'}]P$ podem ser *satisfeitos*, isto é, $\models [\overline{a \mapsto \tau'}]P$ é provável.

A relação de provabilidade de predicados $P \models Q$ é determinada pelo conjunto de declarações de classe e declarações de instância presentes no programa, de acordo com as regras especificadas na Figura 3.2.

Note que a relação $P \models Q$ é parametrizada por um contexto de classe Γ^{c}, o qual é determinado pelas declarações de classes e declarações de instância presentes no programa, e contém dois tipos de termos:

$\mathit{class}\ P \Rightarrow \kappa$	corresponde à primeira linha de uma declaração de classe; cada classe em P é uma superclasse da classe nomeada em κ
$\mathit{inst}\ P \Rightarrow \kappa$	corresponde à primeira linha de uma declaração de instância; se existe uma instância de cada classe em P então existe uma instância de κ

Por Exemplo, as declarações de classe e de instância apresentadas no início desta Seção produzem o contexto de classes Γ^{c} a seguir e introduzem, no contexto de tipos global, digamos Γ_0, uma atribuição de tipo para cada um dos símbolos sobrecarregados

Monotonicidade	(id)	$P \models P$
	(term)	$P \models \emptyset$
	(fst)	$P, Q \models P$
	(snd)	$P, Q \models Q$
	(univ)	$\dfrac{P \models Q \quad P \models R}{P \models Q, R}$
Transitividade	(trans)	$\dfrac{P \models Q \quad Q \models R}{P \models R}$
Fecho	(subs)	$\dfrac{P \models Q}{S\,P \models S\,Q}$
Estruturais	(dist)	$\dfrac{P \models Q \quad P' \models Q'}{P, P' \models Q, Q'}$
	(cut)	$\dfrac{P \models Q \quad P, Q \models R}{P \models R}$

Figura 3.3. Propriedades de $P \models Q$

declarados nas classes:

$$\begin{aligned}\Gamma^{\mathtt{c}} \;=\; & \{\mathit{class}\ \emptyset \Rightarrow \mathtt{Eq}\ a,\ \mathit{class}\ \{\mathtt{Eq}\ a\} \Rightarrow \mathtt{Ord}\ a,\\ & \mathit{inst}\ \emptyset \Rightarrow \mathtt{Eq\ Int},\ \mathit{inst}\ \{\mathtt{Eq}\ a \Rightarrow \{\mathtt{Eq\ [}a\mathtt{]}\}\ \}\end{aligned}$$

$$\begin{aligned}\Gamma \;=\; & \{(==) :: \forall a.\, \mathtt{Eq}\ a \Rightarrow a \to a \to \mathtt{Bool},\\ & (>) :: \forall a.\, \mathtt{Ord}\ a \Rightarrow a \to a \to \mathtt{Bool},\\ & (>=) :: \forall a.\, \mathtt{Ord}\ a \Rightarrow a \to a \to \mathtt{Bool}\ \}\end{aligned}$$

A relação de provabilidade de predicados $P \models Q$ satisfaz as propriedades apresentadas na Figura 3.3, conforme definido em [21].

Dado o contexto de classe $\Gamma^{\mathtt{c}}$ acima, seria possível provar, por Exemplo, $\emptyset \models \{\mathtt{Eq\ [Int]}\}$, usando a regra de transitividade, já que é possível provar, nesse contexto de classe, $\emptyset \models \{\mathtt{Eq\ Int}\}$ e $\{\mathtt{Eq\ Int}\} \models \{\mathtt{Eq\ [Int]}\}$, como se mostra a seguir:

$$\dfrac{\dfrac{}{\emptyset \models \emptyset}\ (id) \qquad (inst\ \emptyset \Rightarrow \texttt{Eq Int}) \in \Gamma^{\texttt{c}}}{\emptyset \models \{\texttt{Eq Int}\}}\ (inst)$$

$$\dfrac{\dfrac{\dfrac{}{\{\texttt{Eq } a\} \models \{\texttt{Eq } a\}}\ (id) \qquad (inst\ \{\texttt{Eq } a\} \Rightarrow \texttt{Eq } [a]) \in \Gamma^{\texttt{c}}}{\{\texttt{Eq } a\} \models \{\texttt{Eq } [a]\}}\ (inst)}{\{\texttt{Eq Int}\} \models \{\texttt{Eq [Int]}\}}\ (subs)$$

3.3 Sistema de Tipos

O sistema de tipos `CTH` é apresentado na Figura 3.4, na forma de regras para julgamentos da forma

$$P; \Gamma \vdash t : \tau$$

significando que "o tipo τ pode ser atribuído ao termo t em um contexto de tipo Γ, se os predicados de classe P podem ser satisfeitos". Para maior brevidade, optamos por apresentar o sistema diretamente em uma versão dirigida por sintaxe, que será mais útil para a posterior definição da inferência de tipos.

As regras de derivação de tipos são semelhantes às do sistema `HM`, mas é requerido aqui um tratamento apropriado do contexto de predicados de classe. Na regra `(APP)`, os predicados de classe que restringem tanto o tipo da função (t) como o tipo do argumento (u) devem ser considerados como restrições sobre o tipo da aplicação ($t\ u$).

Na regra de generalização, os predicados de classe P requeridos na derivação de tipo $P; \Gamma \vdash t : \tau$ são movidos do contexto de predicados de classe para o tipo, e as variáveis livres do tipo resultante $P \Rightarrow \tau$ são quantificadas.

A instanciação é restringida pelos predicados de classe do tipo, em conformidade com a relação de derivabilidade de predicados $P \models Q$. Por, Exemplo, no contexto de classes $\Gamma^{\texttt{c}}$, o tipo $\forall a.\, \texttt{Eq}\ a \Rightarrow a \rightarrow a$ pode ser instanciado, por Exemplo, para `[Int]` $\rightarrow$ `[Int]` por meio da substituição $[a \mapsto \texttt{[Int]}]$, já que $\emptyset \models \texttt{Eq [Int]}$, como provamos anteriormente.

A restrição imposta na derivação do julgamento $\emptyset; \Gamma \vdash t : \sigma$ é usada para evitar a derivação de *tipos ambíguos*. Para entender essa noção de ambiguidade, considere as seguintes definições de classe, que especificam os tipos das funções sobrecarregadas `show` e `read`, para conversão de valores de um determinado tipo a em um valor de tipo `String` e vice-versa, respectivamente:

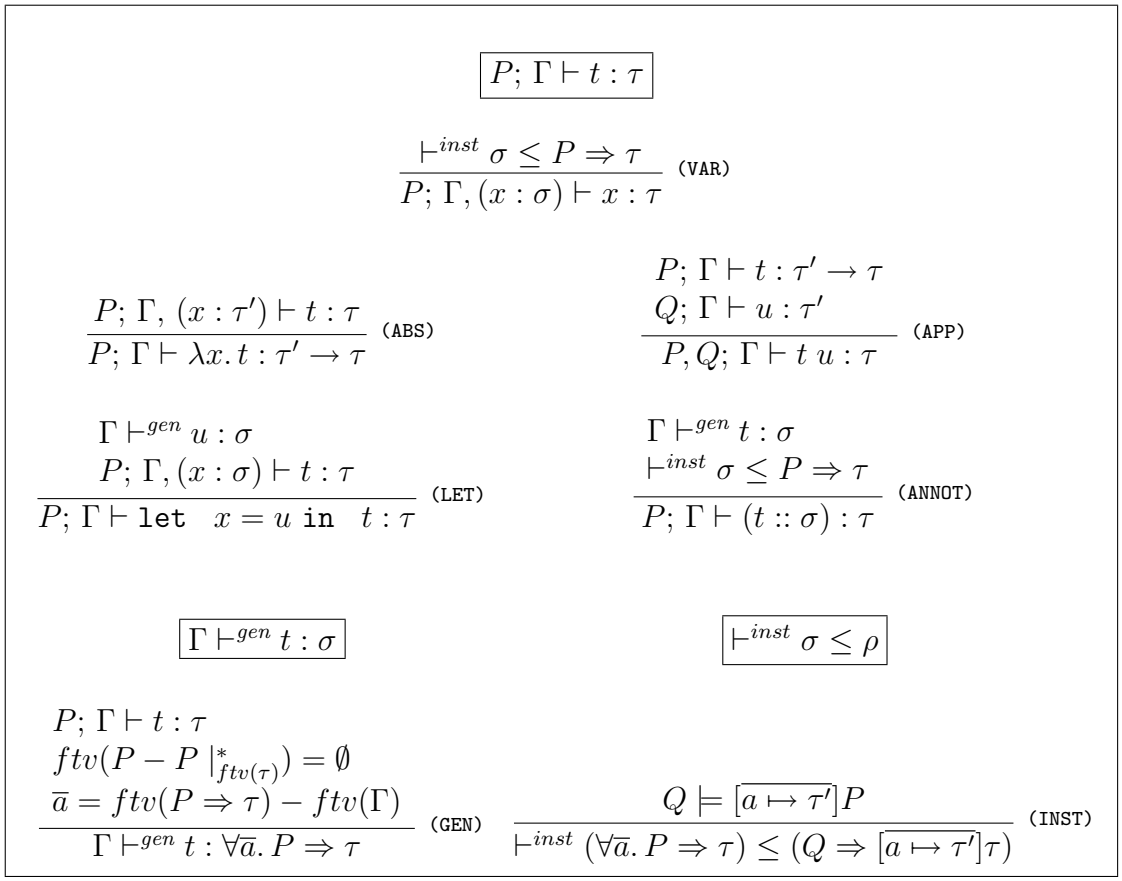

Figura 3.4. Sistema de tipos CTH

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

Na presença dessas declarações de classe, é possível derivar para `show` e `read` os tipos:

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

Nesse contexto seria possível derivar para a expressão `\x-> show(read x)` o tipo $\forall a.$ `(Show` a`,Read` a`)` $\Rightarrow$ `String` $\to$ `String`. Entretanto, esse termo é ambíguo, pois não seria possível determinar quais as instâncias (ou definições) de `read` e `show` deveriam ser usadas nesse caso; o tipo $\forall a.$ `(Show` a`,Read` a`)` $\Rightarrow$ `String` $\to$ `String` é também dito *ambíguo*.

A derivação de tipos ambíguos pode ser evitada impondo-se uma restrição sin-

tática apropriada sobre os tipos deriváveis no sistema de tipos. Para sistemas com suporte a classes de tipos com múltiplos parâmetros essa restrição pode ser expressa como no sistema de tipos `CTH`, tendo sido essa restrição originalmente proposta em [3]. Informalmente, um tipo $\forall a.\, P \Rightarrow \tau$ é *ambíguo* se existe alguma variável em P que não é *atingível* a partir do conjunto de variáveis de τ. Uma variável a de P é atingível a partir de um conjunto de variáveis V se $a \in V$ ou a ocorre em algum predicado $\kappa \in P$ que contém uma variável atingível a partir de V. Uma restrição é atingível a partir de V se contém alguma variável atingível a partir de V. Essa noção é formalizada por meio da operação $P\,|^*_V$, definida na Figura 3.5.

$$P\,|_V = \{C\,\overline{\tau} \in P \mid ftv(\overline{\tau}) \cap V \neq \emptyset\}$$

$$P\,|^*_V = \begin{cases} P\,|_V & \text{if } ftv(P\,|_V) \subseteq V \\ P\,|^*_{ftv(P|_V)} & \text{otherwise} \end{cases}$$

Figura 3.5. Restrições de P atingíveis a partir de V

3.4 Inferência de Tipos

O algoritmo de inferência de tipos para o sistema `CTH` é apresentado na Figura 3.6.

O algoritmo de inferência de tipos é apresentado na forma de um conjunto de regras de derivação para julgamentos da forma

$$P; \Gamma \vdash t : (\rho, P', \Gamma')$$

significando que a tipagem (P', Γ', ρ) é inferida para o termo t no contexto de tipos Γ, e em presença das restrições determinadas pelo contexto de predicados de classe P.

O algoritmo de inferência de tipos é essencialmente o mesmo que para o sistema `HM`, exceto pelo tratamento dos contextos de predicados de classe.

Na regra da aplicação, o contexto de predicados inferido para a aplicação $(t\ u)$ é obtido pela união dos contextos de predicados de classe usados na inferência de tipos para t e para u. Para entender a razão disso, considere a inferência de tipos para a aplicação $(t\ u)$, em um contexto Γ, onde $\Gamma = \{t : \forall a.\, (\texttt{T}\ a) \Rightarrow (a \to \texttt{Int}) \to a,\ u :$

$$\boxed{P;\ \Gamma \vdash t : (\tau, P', \Gamma')}$$

$$\frac{\vdash^{inst} \sigma \leq P \Rightarrow \tau}{P;\ \Gamma, (x : \sigma) \vdash x : (\tau, P, \Gamma)}\ \texttt{(VAR)}$$

$$\frac{\begin{array}{l} P;\ \Gamma_x,\ (x : a) \vdash t : (\tau, P', \Gamma') \\ a\ \textit{fresh} \end{array}}{P;\ \Gamma \vdash \lambda x.\, t : (\Gamma'(x) \to \tau, P', \Gamma'_{[x,\Gamma]})}\ \texttt{(ABS)} \qquad \frac{\begin{array}{l} P;\ \Gamma \vdash t : (\tau_1, P_1, \Gamma_1) \\ P;\ \Gamma_1 \vdash u : (\tau_2, P_2, \Gamma_2) \\ S = \mathit{unify}(\tau_1 = \tau_2 \to a) \\ a\ \textit{fresh} \end{array}}{P;\ \Gamma \vdash t\ u : (Sa, S(P_1, P_2), S\Gamma_2)}\ \texttt{(APP)}$$

$$\frac{\begin{array}{l} \Gamma \vdash^{gen} u : (\sigma, \Gamma'') \\ P;\ \Gamma''_x, (x : \sigma) \vdash t : (\tau, P', \Gamma') \end{array}}{P;\ \Gamma \vdash \texttt{let}\ \ x = u\ \texttt{in}\ \ t : (\tau, P', \Gamma'_{[x,\Gamma]})}\ \texttt{(LET)} \qquad \frac{\begin{array}{l} \Gamma \vdash^{gen} t : (\sigma', \Gamma') \\ \vdash^{sh} \sigma' \leq \sigma \\ \vdash^{inst} \sigma \leq P' \Rightarrow \tau \end{array}}{P;\ \Gamma \vdash (t :: \sigma) : (\tau, P', \Gamma')}\ \texttt{(ANNOT)}$$

$$\boxed{\Gamma \vdash^{gen} t : (\sigma, \Gamma')} \qquad \boxed{\vdash^{inst} \sigma \leq \rho}$$

$$\frac{\begin{array}{l} P;\ \Gamma \vdash t : (\tau, Q, \Gamma') \\ ftv(Q - Q\mid^*_{ftv(\tau)}) = \emptyset \\ \overline{a} = ftv(Q \Rightarrow \tau) - ftv(\Gamma') \end{array}}{\Gamma \vdash^{gen} t : (\forall \overline{a}.\, Q \Rightarrow \tau, \Gamma')}\ \texttt{(GEN)} \qquad \frac{\begin{array}{l} Q \models [\overline{a \mapsto b}]P \\ \overline{b}\ \textit{fresh} \end{array}}{\vdash^{inst} \forall \overline{a}.\, P \Rightarrow \tau \leq Q \Rightarrow [\overline{a \mapsto b}]\tau}\ \texttt{(INST)}$$

$$\boxed{\vdash^{sh} \sigma \leq \sigma'}$$

$$\frac{\begin{array}{l} S = \mathit{match}(\tau \rhd \tau') \\ Q \models SP \\ \overline{b}\ \textit{fresh} \end{array}}{\vdash^{sh} \forall \overline{a}.\, P \Rightarrow \tau \leq \forall \overline{b}.\, Q \Rightarrow \tau'}\ \texttt{(SH)}$$

Figura 3.6. Inferência de tipos para `CTH`

$\forall a.\, (\texttt{U}\ \ a) \Rightarrow a \to a\}$ e um contexto de predicados vazio, a qual é apresentada a seguir:

$$\frac{\vdash^{inst} \forall a.\, (\texttt{T}\ \ a) \Rightarrow (a \to \texttt{Int}) \to a \leq (\texttt{T}\ \ a_1) \Rightarrow (a_1 \to \texttt{Int}) \to a_1}{(\texttt{T}\ \ a_1);\ \Gamma \vdash t : ((a_1 \to \texttt{Int}) \to a_1, (\texttt{T}\ \ a_1), \Gamma)}\ \texttt{(VAR)}$$

$$\frac{\vdash^{inst} \forall a.\, (\texttt{U}\ \ a) \Rightarrow a \to a \leq (\texttt{U}\ \ a_2) \Rightarrow a_2 \to a_2}{(\texttt{U}\ \ a_2);\ \Gamma \vdash u : (a_2 \to a_2, (\texttt{U}\ \ a_2), \Gamma)}\ \texttt{(VAR)}$$

$$\frac{S = \mathit{unify}((a_1 \to \texttt{Int}) \to a_1 = (a_2 \to a_2) \to a) \quad a\ \textit{fresh}}{\emptyset;\ \Gamma \vdash (t\ u) : (\texttt{Int}, (\texttt{G}\ \ \texttt{Int}, \texttt{O}\ \ \texttt{Int}), \Gamma)}\ \texttt{(APP)}$$

As regras que definem a relação de ordem em tipos polimórficos $\vdash^{sh} \sigma' \leq \sigma$ são também modificadas levando em conta a relação de provabilidade de predicados. Note que as relações $\vdash^{inst} \sigma \leq \rho$ e $\vdash^{sh} \sigma' \leq \sigma$ são implicitamente parametrizadas pelo contexto de classe Γ^{c}, já que este induz a relação $P \models Q$.

Para que o sistema apresentado na Figura 3.6 seja de fato um algoritmo, deve ser ainda definido um algoritmo para implementação da relação de provabilidade de predicados $P \models Q$. Vamos omitir aqui a definição desse algoritmo, remetendo o leitor interessado aos trabalhos apresentados, por Exemplo, em [4, 48]. É preciso aqui observar que, para que tal algoritmo exista, isto é, para que a relação $P \models Q$ seja decidível, é necessário impor restrições apropriadas sobre a forma das declarações de classes e declarações de instâncias, uma vez que a relação $P \models Q$, para predicados de classe arbitrários, é indecidível (veja [47]).

O algortimo apresentado na Figura 3.6 é correto em relação ao sistema de tipos `CTH` e infere tipos principais para termos desse sistema. Essas propriedades são expressas formalmente pelos teoremas a seguir, cujas provas podem ser encontradas em [21].

Teorema 3.4.1 (Correção) *Se* $P; \Gamma \vdash t : (\tau, P', \Gamma')$ *então* $P'; \Gamma' \vdash t : \tau$

Teorema 3.4.2 (Tipo Principal) *Para todo termo* t*, se* $P; \Gamma \vdash^{gen} t : \sigma$*, então* $P; \Gamma \vdash^{gen} t : (\tau', \Gamma')$ *, onde* $\vdash^{sh} \Gamma' \leq \Gamma$ *e* $\vdash^{sh} \sigma' \leq \sigma$*.*

3.4.1 Semântica

A semântica de classes de tipos de Haskell pode ser definida no chamado estilo de "passagem de dicionário" proposto por Wadler e Blott [53], e que constitui um caso particular da semântica para sistemas de tipos com predicados definida por Mark Jones em [21]. Para compreender essa idéia considere novamente as seguintes declarações de classe e de instância, apresentadas anteriormente na introdução desta Seção:

```
class Eq a where
     (==) ::  a -> a -> Bool
     (/=) ::  a -> a -> Bool
```

```
class Eq a => Ord a where
     (>) ::  a -> a -> Bool
     (>=) ::  a -> a -> Bool

instance Eq Int where
     i1 == i2 = primEqInt i1 i2
     i1 /= i2 = not (i1 == i2)

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

 elem ::  Eq a => a -> [a] -> Bool
 elem x [] = False
 elem x (y:ys) | x==y = True
               | otherwise = elem x ys
```

Informalmente, a semântica de "passagem de dicionário" pode ser compreendida do seguinte modo:

- uma definição de uma classe C é traduzida como um tipo polimórfico de um *dicionário* que contém definições de símbolos cuja assinatura é especificada nessa classe, sendo os parâmetros desse tipo exatamente os parâmetros da classe;

- uma declaração de instância da classe C é traduzida como valor de uma instância do tipo do *dicionário* de C, o qual contém definições específicas para cada símbolo sobrecarregdo da classe C;

- uma expressão que envolve um símbolo sobrecarregada definido em uma classe C é traduzida como uma função que tem um parâmetro adicional, cujo tipo é o tipo do *dicionário* dessa classe.

De acordo com essa semântica, as declarações apresentadas acima, por Exemplo, poderiam ser traduzidas do seguinte modo:

```
  - tipo do dicionário da classe Eq
  data Eq a = DEq {(==)::(a->a->Bool), (/=)::(a->a->Bool)}
  - tipo do dicionário da classe Ord
  data Ord a = DOrd {dEq::Eq a, (>)::a->a->Bool, (>=)::a->a->Bool}


- dicionário correspondente a instância Eq Int
dEqInt ::  Eq Int
dEqInt = DEq {(==) = primEqInt, (/=) = \ x y -> not (x == y)}

- dicionário correspondente a instância Eq [a]
dEqList ::  Eq a -> Eq [a]
dEqList d = DEq {(==) = el, (/=) = \ x y -> not (el x y)}
   where el []     []     = True
         el (x:xs) (y:ys) = d.(==) x y && el xs ys
         el _      _      = False

- dicionário correspondente à instância Ord Int
dOrdInt ::  Ord Int
dOrdInt = DOrd {dEq = dEqInt, (>) = primGtInt,
                (>=) = \ x y -> (x primGtInt y) || (dEq.(==) x y)}

- elem tem como parâmetro adicional um dicionário da classe Eq
elem ::  Eq a => a -> [a] -> Bool
elem d x [] = False
elem d x (y:ys) | d.(==) x y = True
                | otherwise = elem d x ys
```

Uma definição formal da semântica de expressões do sistema `CTH` pode ser encontrada em [21, 15], onde a semântica é definida na forma de uma tradução (dirigida por sintaxe) de expressões do sistema `CTH` para expressões bem tipadas do sistema `F`.

3.5 Conclusão

Este Capítulo apresentou uma revisão sobre o mecanismo de suporte a sobrecarga baseado de Haskell, em classes de tipos, apresentado um sistema de tipos que formaliza esse mecanismo, o algoritmo de inferência de tipos correspondente e a semântica de

expressões nesse sistema de tipos, definida no estilo de "passagem de dicionário".

Nesse sistema, é possível definir, por Exemplo, uma função `sort`, para ordenar elementos de uma lista (digamos em ordem não decrescente), com tipo `sort` :: $\forall a.$ `(Ord` a`)` $\Rightarrow [a] \rightarrow [a]$, significando que tal função pode ser aplicada a listas de elementos de qualquer tipo a que seja instância da classe `Ord`, ou seja, qualquer tipo a para o qual as relações de comparação `(>)` e `(>=)` sejam definidas.

Fazendo uso mais uma vez do Exemplo 1.1 (3), seria bastante razoável esperar que fosse válida a aplicação `foo sort`, resultando no valor `([False,True],['a','b',''c])`. Essa aplicação entretanto não é válida, mesmo em uma linguagem com suporte para sobrecarga e para polimorfismo de rank superior, pois essas duas extensões do sistema `HM` não estão integradas. Note que o tipo de `sort` é igual ao do parâmetro de `foo`, a menos da restrição de classe no tipo do primeiro.

É do nosso conhecimento apenas um trabalho que busca resolver essa questão, possibilitando a aplicação de uma função com parâmetro de tipo polimórfico a uma expressão de tipo polimórfico restrito: o sistema `QMLF`, proposto em [33]. Nesse sistema, é possível, por Exemplo, anotar para `foo` o tipo:

$$\texttt{foo} :: (\forall a.\, \texttt{Ord}\ a \Rightarrow [a] \rightarrow [a]) \rightarrow (\texttt{[Bool]}, \texttt{[Char]})$$

Essa anotação de tipo possibilita tanto a aplicação `foo sort`, como a aplicação `foo reverse`, já que, no sistema `QMLF`, o tipo de `reverse`, qual seja, `reverse` :: $\forall a.\, [a] \rightarrow [a]$, pode ser para o tipo polimórfico restrito $\forall a.$ `(Ord` a`)` $\Rightarrow [a] \rightarrow [a]$.

Entretanto, a anotação de tipo acima apenas possibilita aplicar `foo` a um argumento de tipo $\forall a.\, [a] \rightarrow [a]$ ou $\forall a.$ `(Ord` a`)` $\Rightarrow [a] \rightarrow [a]$, não sendo possível, por Exemplo, a aplicação de `foo` a um argumento de tipo $\forall a.$ `(C` a`)` $\Rightarrow [a] \rightarrow [a]$, onde `C` é uma classe distinta de `Ord`.

As questões que buscamos responder neste trabalho podem ser expressas do seguinte modo:

- Que forma anotação de tipo poderia ser provida de modo a permitir a definição de funções tais como `foo`, em que o parâmetro é usado polimorficamente no corpo da função, possibilitando o reuso de tais funções em diferentes contextos?
- Como definir um sistema de tipos em que tal anotação de tipo possa ser especificada?
- Como definir a semântica e a inferência de tipos para expressões desse sistema?

Nossa resposta para essas questões é apresentada no Capítulo 4 a seguir.

Capítulo 4

Funções Polimórficas Restritas de Primeira Classe

Este capítulo apresenta a definição de uma extensão do sistema de suporte a sobrecarga `CTH`, apresentado no capítulo 3. Essa extensão — doravante denominada `CTi` — possibilita a definição de funções com parâmetros de tipo polimórfico, mediante anotação explícita do tipo de tais funções pelo programador, sendo esses tipos especificados na forma de tipos interseção. No sistema proposto, funções com parâmetros polimórficos podem tanto ser aplicadas a argumentos de tipo polimórfico paramétrico (como em sistemas de polimorfismo de rank superior), como a argumentos de tipo polimórfico restrito, promovendo valores sobrecarregados a objetos de primeira classe.

A Seção 4.1 revê a motivação para o sistema `CTi` e descreve intuitivamente as idéias básicas desse sistema. A Seção 4.2 apresenta a sintaxe de tipos e expressões do sistema `CTi`, sendo as regras de derivação de tipos nesse sistema apresentadas na Seção 4.3. A inferência de tipos é apresentada na Seção 4.4.

4.1 Introdução

Como poderia ser especificado o tipo de uma função cujo argumento é usado polimorficamente no corpo da sua definição? Naturalmente, cada argumento é usado no corpo de uma função apenas um número finito de vezes, e gostaríamos de permitir que em cada uso o seu tipo pudesse ser diferente, desde que, em cada caso, o tipo com que o argumento é usado seja uma "instância" do tipo especificado para esse argumento na anotação de tipo. O uso de uma *tipo interseção* nos parece natural para expressar esse requerimento.

Tipos interseção foram propostos por Coppo, Dezani-Ciancaglini e Venneri [8],

como uma alternativa ao sistema HM para estender o lambda-calculus tipado simples, utilizando polimorfismo finitário em lugar de polimorfismo infinitário, ou universal, como no sistema HM. Os dois sistemas não são equivalentes. Existem expressões que não podem ser tipadas em HM e que podem ser tipadas em um sistema de tipos interseção e vice-versa. Por exemplo, a expressão $\lambda x.\, xx$, à qual pode ser atribuído o tipo $(a \wedge (a \to b)) \to b$. Por outro lado, em um sistema de tipos interseção não é possível expressar, por exemplo, um tipo para o operador de igualdade entre listas, de modo que esse operador possa ser usado para comparar listas de elementos de um tipo qualquer, o que é possível em uma extensão do sistema HM com polimorfismo restrito, tal como explicado anteriormente.

Diferentemente de sistemas usuais de tipos interseção, no sistema de tipos definido neste trabalho não se pretende utilizar polimorfismo finitário em lugar de polimorfismo infinitário. Ao contrário, o sistema aqui proposto busca manter mantém total compatibilidade com o sistema HM na ausência de anotações de tipo, utilizando tipos interseção apenas para estender HM de maneira que seja possível definir funções nas quais o parâmetro é usado com diferentes tipos no corpo da função. Nesse caso, é requerida uma anotação explícita do tipo desse parâmetro na definição da função.

Intuitivamente, um valor de tipo interseção $\sigma_1 \wedge \sigma_2$ é visto como uma valor que representa objetos de tipo σ_1 e de tipo σ_2. Sendo assim, sistemas de tipo interseção incluem as seguintes regras genéricas de introdução (generalização) e eliminação (instanciação) de tipos interseção:

$$\frac{\Gamma \vdash t : \sigma_1 \quad \Gamma \vdash t : \sigma_2}{\Gamma \vdash t : \sigma_1 \wedge \sigma_2}\ \texttt{(GENi)} \qquad \frac{\Gamma \vdash t : \sigma_1 \wedge \sigma_2}{\Gamma \vdash t : \sigma_1}\ \texttt{(INSTi)} \qquad \frac{\Gamma \vdash t : \sigma_1 \wedge \sigma_2}{\Gamma \vdash t : \sigma_2}\ \texttt{(INSTi)}$$

Sistemas de tipo polimórficos podem ser definidos de forma dirigida por sintaxe, restringindo-se a instanciação de tipos polimórficos à regra (VAR) e a generalização à regra (LET). Em um sistema genérico de tipos interseção, seria necessário a introdução de tipos interseção na conclusão de cada regra assim como eliminação de tipos interseção em cada uma das premissas da regra. Isso resulta em enorme complexidade da inferência de tipos. Por esse motivo, assim como em outros sistemas de tipo interseção [27, 9], no sistema proposto neste trabalho tipos interseção não podem ocorrer do lado direto do construtor de tipos funcional ($\to$), podendo ocorrer apenas como tipos de parâmetros de lambda-abstrações. Mediante essa restrição, é possível limitar a introdução de tipos interseção apenas à regra da aplicação, usando a introdução de interseção para obter o tipo do argumento, quando o tipo do parâmetro da função é um tipo interseção. Dualmente, a eliminação de tipo interseção fica restrita à regra (VAR), sendo isso requerido para derivar um tipo monomórfico a partir de um tipo interseção

exsitente no contexto de tipos, o qual corresponde a uma anotação explícita de tipo interseção para o parâmetro de uma função.

Mesmo impondo a restrição de que tipos interseção não podem ocorrer do lado direito do construtor de tipos funcionais, a inferência de tipos em um sistema de tipos interseção é extremamente complexa (veja, por exemplo, [27]). Essa complexidade é menor no sistema `CTi`, uma vez que tipos interseção apenas são introduzidos por meio de anotações de tipo explícitas.

O sistema de tipos proposto neste trabalho combina de maneira original tipos polimórficos (tanto paramétricos como restritos por predicados de classe) e tipos interseção, usando estes últimos apenas para possibilitar a definição de funções em que o parâmetro é usado com tipos distintos no corpo da sua definição.

Como mencionamos na introdução desse trabalho, no sistema `CTi` é possível definir a função `foo` do exemplo 1.1 (3), provendo, por exemplo, a anotação de tipo

$$\texttt{foo} :: \forall a, b.\, (\texttt{[Bool]} \to a \wedge \texttt{[Char]} \to b) \to (a, b)$$

Essa anotação de tipo é suficientemente expressiva para possibilitar a aplicação de `foo` a qualquer dos argumentos considerados anteriormente (seja aqueles com tipo polimórfico paramétrico ou com tipo polimórfico restrito), em geral a qualquer expressão cujo tipo possa ser instanciado tanto para `[Bool]` $\to a$ como para `[Char]` $\to b$.

4.2 Sintaxe

A linguagem de tipos e expressões do sistema `CTi` é apresentada na Figura 4.1. Parâmetros de funções podem ser agora uma interseção de tipos monomórficos (veja a definição de tipos denotados pela metavariável ι). Escrevemos $\wedge\overline{\tau}$ para representar um tipo interseção $(\tau_1 \wedge \ldots \wedge \tau_n)$, para $n \geq 1$, e supomos que o construtor de tipos interseção ($\wedge$) satisfaz as seguintes propriedades:

Idempotência	$\tau \wedge \tau = \tau$
Comutatividade	$\tau_1 \wedge \tau_2 = \tau_2 \wedge \tau_1$
Associatividade	$\tau_1 \wedge (\tau_2 \wedge \tau_3) = (\tau_1 \wedge \tau_2) \wedge \tau_3$

Escrevemos $\tau \in (\tau_1 \wedge \ldots \wedge \tau_n)$ se $\tau = \tau_i$ para algum $1 \leq i \leq n$ e $\rho \subseteq (\tau_1 \wedge \ldots \wedge \tau_n)$ se $\tau \in \rho$ implica $\tau \in (\tau_1 \wedge \ldots \wedge \tau_n)$.

A definição de $ftv(\sigma)$, que computa as variáveis livres de um tipo σ, é naturalmente estendida de modo a levar em conta tipos interseção, definindo-se que $ftv(\tau_1 \wedge$

Tipos	τ	::=	$a \mid \iota \to \tau$
Tipos interseção	ι	::=	$\wedge\overline{\tau}$
Tipos com restrições	ρ	::=	$P \Rightarrow \tau$
Tipos polimórficos	σ	::=	$\forall\overline{a}.\,\rho$
Predicado de classes	κ	::=	$C\,\overline{\tau}$
Contexto de predicados	P, Q	::=	$\epsilon, \mid P, \kappa$
Contexto de tipos	Γ	::=	$\epsilon \mid \Gamma, x : \sigma \mid \Gamma, (x : \iota)$
Contexto de classes	$\Gamma^{\mathtt{c}}$	::=	$\epsilon \mid \Gamma^{\mathtt{c}}, (\mathit{class}\ P \Rightarrow \kappa) \mid \Gamma^{\mathtt{c}}, (\mathit{inst}\ P \Rightarrow \kappa)$
Programas	p	::=	$\overline{cD};\ \overline{iD};\ t$
Declaração de classe	cD	::=	$\mathtt{class}\ P \Rightarrow C\,\overline{\tau}\ \mathtt{where}\ \overline{x :: \tau}$
Declaração de instância	iD	::=	$\mathtt{instance}\ P \Rightarrow C\,\overline{\tau}\ \mathtt{where}\ \overline{x = e}$

Termos	t, u	::=	x	Variável
		$\mid$	$\lambda x.\,t$	Abstração funcional
		$\mid$	$\lambda(x :: \rho).\,t$	Abstração com anotação de tipo interseção
		$\mid$	$t\ u$	Aplicação
		$\mid$	$\mathtt{let}\ x = u\ \mathtt{in}\ \ t$	Definição local
		$\mid$	$t :: \sigma$	Anotação de tipo (σ fechado)

Figura 4.1. Sintaxe da linguagem do sistema `CTi`

$\ldots\wedge\tau_n) = ftv(\tau_1)\cup\ldots\cup ftv(\tau_1)$. A aplicação de substituição $[\overline{a \mapsto \tau}]\sigma$ é também estendida apropriadamente, definindo-se $[\overline{a \mapsto \tau}](\tau_1 \wedge \ldots \wedge \tau_n) = [\overline{a \mapsto \tau}]\tau_1 \wedge \ldots \wedge [\overline{a \mapsto \tau}]\tau_n$.

Tal como no sistema `CTH` (veja 3.1), um programa consiste de declarações de classe e declarações de instância, seguidas de uma expressão. Tipos polimórficos podem incluir um conjunto (possivelmente vazio) de restrições, na forma de predicados de classe, e um tipo funcional pode ter, do lado esquerdo de ($\to$), um tipo interseção. Expressões são também como no sistema `CTH` exceto que agora consideramos também lambda-abstrações com anotação explícita do tipo do parâmetro, na forma $\lambda(x : \rho).\,t$.

Note que um contexto de tipos (Γ) pode conter tanto tipos polimórficos (σ), como tipos interseção (ι).

4.3 Sistema de Tipos

O sistema de tipos `CTi` é apresentado na Figura 4.2, na forma dirigida por sintaxe, sendo definido como um conjunto de regras de derivação para julgamentos da forma

$$P; \Gamma \vdash t : \rho$$

significando que "o tipo ρ pode ser atribuído ao termo t em um contexto de tipos Γ, se os predicados de classe P podem ser satisfeitos".

Como no sistema `CTH` (veja 3.4), a derivação de tipos é parametrizada pela relação de prova de predicados de classe $P \models Q$, induzida pelas declarações de classe e declarações de instância presentes no programa.

O sistema de tipos `CTi` difere do sistema `CTH` apenas nas regras `(ABSA)`, `(APP)` e `(VAR)`. A regra `(ABSA)` define a derivação de tipos para uma lambda-abstração com anotação explícita do tipo do parâmetro, de maneira semelhante à regra `(ABS)`.

A regra `(APP)` deve considerar agora duas possibilidades inexistentes no sistema `CTH`, para uma aplicação $(t\ u)$:

- A possibilidade do tipo do parâmetro da função (f) ser um tipo interseção $(\wedge\overline{\tau})$. Nesse caso, o tipo do argumento (u) deve poder ser "promovido" para $(\wedge\overline{\tau})$, o que significa que o tipo de (u) deve poder ser instanciado para cada um dos tipos $\tau \in \overline{\tau}$.

- A possibilidade de ser necessária uma eliminação de tipo interseção, para obtenção do tipo da função (t). Isso ocorre apenas se a aplicação $(t\ u)$ ocorre no corpo de uma função, e t é um parâmetro dessa função, cujo tipo é uma interseção $(\wedge\overline{\tau'})$. Portanto, nesse caso, t é uma variável, sendo o tipo de t obtido diretamente a partir do contexto de tipos.

O primeiro caso acima é tratado por meio do julgamento de tipo auxiliar

$$P; \Gamma \vdash^{\wedge_I} t : \iota$$

que "promove" o tipo do argumento para um tipo interseção, se for o caso, usando introdução de tipo interseção. O segundo caso é tratado por meio do julgamento

$$P; \Gamma^{\wedge_E} t : \tau$$

As demais regras de derivação são tais como no sistema `CTH`, sendo também como anteriormente as regras para derivação dos julgamentos $\Gamma \vdash^{gen} t : \sigma$, $\vdash^{inst} \sigma \Rightarrow \tau$ e

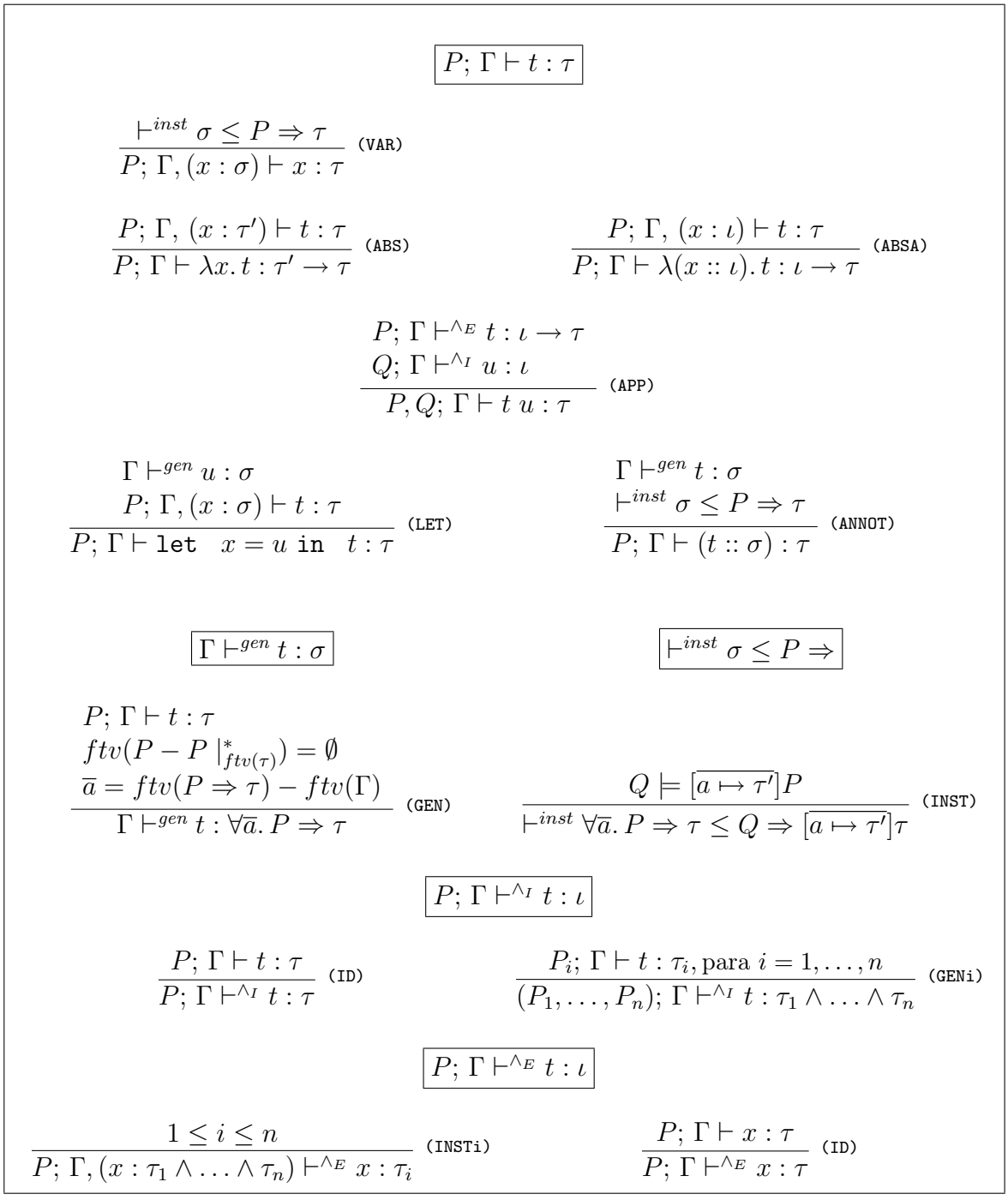

Figura 4.2. Sistema de tipos `CTi`

($\vdash^{sh} \sigma : \sigma'$) (veja Figura 3.4).

Para melhor compreender as regras de derivação do sistema `CTi`, considere a derivação de tipo para a aplicação `foo sort`, sendo `foo` a função definida no exemplo 1.1

(3), em um contexto Γ, em que os tipos de `foo` e de `sort` são:

$$\sigma_{\texttt{foo}} = \forall a, b.\,(\texttt{[Bool]} \;\rightarrow\; a \;\wedge\; \texttt{[Char]} \;\rightarrow\; b) \;\rightarrow\; (a, b)$$
$$\sigma_{\texttt{sort}} = \texttt{sort} :: \forall a.\,\texttt{(Ord}\ a\texttt{)} \Rightarrow [a] \rightarrow [a]$$

Na regra `(APP)`, a descrição do tipo da função `foo` é como a seguir:

$$\dfrac{\dfrac{\vdash^{inst} \sigma_{\texttt{foo}} \leq ((\texttt{[Bool]} \rightarrow \texttt{[Bool]}) \wedge (\texttt{[Char]} \rightarrow \texttt{[Char]})) \rightarrow (\texttt{[Bool]}, \texttt{[Char]})}{\emptyset;\, \Gamma, (\texttt{foo} : \sigma_{\texttt{foo}}) \vdash \texttt{foo} : ((\texttt{[Bool]} \rightarrow \texttt{[Bool]}) \wedge (\texttt{[Char]} \rightarrow \texttt{[Char]})) \rightarrow (\texttt{[Bool]}, \texttt{[Char]})}\ \texttt{(VAR)}}{\emptyset;\, \Gamma, \vdash^{\wedge_E} \texttt{foo} : ((\texttt{[Bool]} \rightarrow \texttt{[Bool]}) \wedge (\texttt{[Char]} \rightarrow \texttt{[Char]})) \rightarrow (\texttt{[Bool]}, \texttt{[Char]})}\ \texttt{(ID)}$$

O tipo de `sort` é "promovido" para o tipo $((\texttt{[Bool]} \rightarrow \texttt{[Bool]}) \wedge (\texttt{[Char]} \rightarrow \texttt{[Char]}))$, por meio da derivação de tipo a seguir:

$$\dfrac{\vdash^{inst} \sigma_{\texttt{sort}} \leq \texttt{(Ord [Bool])} \Rightarrow \texttt{[Bool]} \rightarrow \texttt{[Bool]}}{\texttt{(Ord [Bool])};\, \Gamma, (\texttt{sort} : \sigma_{\texttt{sort}}) \vdash \texttt{sort} : \texttt{[Bool]} \rightarrow \texttt{[Bool]}}\ \texttt{(VAR)}$$

$$\dfrac{\vdash^{inst} \sigma_{\texttt{sort}} \leq \texttt{(Ord [Char])} \Rightarrow \texttt{[Char]} \rightarrow \texttt{[Char]}}{\texttt{(Ord [Char])};\, \Gamma, (\texttt{sort} : \sigma_{\texttt{sort}}) \vdash \texttt{sort} : \texttt{[Char]} \rightarrow \texttt{[Char]}}\ \texttt{(VAR)}$$

$$\dfrac{}{\texttt{(Ord [Bool],Ord [Char])};\, \Gamma \vdash^{\wedge_I} \texttt{sort} : (\texttt{[Bool]} \rightarrow \texttt{[Bool]}) \wedge (\texttt{[Char]} \rightarrow \texttt{[Char]})}\ \texttt{(GENi)}$$

Usando as conclusões das derivações acima, obtemos finalmente, na regra `(APP)`:

$$\dfrac{\begin{array}{l}\emptyset;\, \Gamma \vdash^{\wedge_E} \texttt{foo} : ((\texttt{[Bool]} \rightarrow \texttt{[Bool]}) \wedge (\texttt{[Char]} \rightarrow \texttt{[Char]})) \rightarrow (\texttt{[Bool]}, \texttt{[Char]}) \\ \texttt{(Ord [Bool],Ord [Char])};\, \Gamma \vdash^{\wedge_I} \texttt{sort} : ((\texttt{[Bool]} \rightarrow \texttt{[Bool]}) \wedge (\texttt{[Char]} \rightarrow \texttt{[Char]}))\end{array}}{\texttt{(Ord [Bool],Ord [Char])};\, \Gamma \vdash \texttt{foo sort} : \texttt{([Bool],[Char])}}\ \texttt{(APP)}$$

O tipo derivado para `foo sort` é `([Bool],[Char])`, devendo ser observado que, nessa derivação de tipos, o contexto de classe $\Gamma^{\texttt{c}}$ deve ser tal que a relação de provabilidade de predicados induzida por esse contexto prova $\emptyset \models \{\texttt{Ord [Bool]}\}$ e $\emptyset \models \{\texttt{Ord [Char]}\}$.

Considere agora a derivação do tipo de `g` na aplicação `g [True,False]`, no corpo da função `foo`. O tipo de `g` é derivado do seguinte modo:

$$\dfrac{}{\emptyset;\, \Gamma, (\texttt{g} : \sigma_{\texttt{g}}) \vdash^{\texttt{(INSTi)}} \texttt{g} : \texttt{[Bool]} \;\rightarrow\; a}\ \texttt{(GENi)}$$

Considere ainda uma expressão da forma $\backslash(x :: \wedge\overline{\tau}).\, t$, para a qual $(\wedge\overline{\tau})$ consiste na conjunção de dois tipos, tais como, por exemplo, $\tau \rightarrow \tau'_1$ e $\tau \rightarrow \tau'_2$ e tal que o parâmetro (x) é aplicado a um argumento de tipo τ, no corpo (t) dessa função. Nesse caso, existem duas possíveis escolhas para o tipo do parâmetro (x) quando usado nessa aplicação: $\tau \rightarrow \tau'_1$ ou $\tau \rightarrow \tau'_2$; isso significa que seria possível derivar tanto o tipo τ'_1 como τ'_2

para essa aplicação. Isso caracteriza uma situação de ambiguidade: não seria possível determinar qual das implementações ligadas a x deveria ser usada nessa aplicação — a implementação de tipo $\tau \to \tau_1'$ ou a implementação de tipo $\tau \to \tau_2'$. Veremos, na Seção 4.4, que é possível detectar essa situação de ambiguidade na inferência de tipos, podendo ser reportada, nesse caso, uma mensagem de erro de "anotação de tipo ambíguo".

É claro que não é possível obter uma derivação de tipo para uma função da forma $\backslash(x :: \wedge\overline{\tau}).\, t$, se o parâmetro x é usado no corpo da função com algum tipo (monomórfico) que não ocorre em $(\wedge\overline{\tau})$. Por outro lado, as regras do sistema de tipos não impedem que uma anotação $(\wedge\overline{\tau})$ inclua algum tipo τ' que não corresponde a nenhum dos usos do parâmetro x no corpo da função. Pode ser interessante também considerar inválidas anotações de tipo tais como essa, mas para isso o sistema de tipos teria que ser modificado de modo a propagar, ao longo das derivações de tipo, alguma informação sobre os tipos de cada um dos usos, no corpo de uma função, para um parâmetro dessa função que tenha tipo interseção.

4.4 Inferência de Tipos

O algoritmo de inferência de tipos para `CTi` é apresentado na Figura 4.3. Ele é semelhante ao algoritmo de inferência de tipos para o sistema `CTH` (veja Figura 3.6), diferindo entretanto nas regras `(ABSA)`, `(APP)` e `(VAR)`, tal como ocorre para o sistema de tipos.

No algoritmo de inferência de tipos existem duas regras para inferência de tipos para uma aplicação $(t\ u)$, correspondendo aos seguintes possíveis casos, que devem ser considerados:

- A regra `(APP1)` é usada no caso em que o termo t é uma variável (x). Nesse caso é necessário considerar a possibilidade de que o tipo de t seja um tipo interseção, o que é possível já que tipos interseção podem ocorrer no contexto de tipos. Se for esse o caso, esse tipo interseção deve ser "eliminado", de maneira apropriada, selecionando-se um dentre os tipos que constituem essa interseção. Essa seleção é realizada conforme o tipo do argumento da aplicação, sendo por essa razão necessário inferir primeiro o tipo do argumento, e depois o tipo da função, nessa forma da regra da aplicação. Se nenhum, ou mais de um, dentre os tipos que compõem o tipo interseção da função puder ser selecionado, conforme requerido pelo tipo do argumento, é reportado um "erro de tipo". Portanto, na inferência de tipos é possível detectar uma situação de uso ambíguo de um

$$\boxed{P;\,\Gamma \vdash t : (\tau, P', \Gamma')}$$

$$\frac{\vdash^{inst} \sigma \leq P' \Rightarrow \tau}{P;\,\Gamma, (x:\sigma) \vdash x : (\tau, P', \Gamma)}\ \texttt{(VAR)}$$

$$\frac{\begin{array}{l} P;\,\Gamma,\,(x:a) \vdash t : (\tau, P', \Gamma') \\ a\ \textit{fresh} \end{array}}{P;\,\Gamma \vdash \lambda x.\, t : (\Gamma'(x) \to \tau, P', \Gamma'_{[x,\Gamma]})}\ \texttt{(ABS)}$$

$$\frac{\begin{array}{l} P;\,\Gamma,\,(x:\iota) \vdash t : (\tau, P', \Gamma') \\ a\ \textit{fresh} \end{array}}{P;\,\Gamma \vdash \lambda(x :: \iota).\, t : (\iota \to \tau, P', \Gamma'_{[x,\Gamma]})}\ \texttt{(ABSA)}$$

$$\frac{\begin{array}{l} P;\,\Gamma \vdash u : (\tau, P_2, \Gamma_2) \\ P;\,\Gamma_2 \vdash^{\wedge_E} [\tau]x : (P_1, \Gamma_1, S, \tau') \end{array}}{P;\,\Gamma \vdash x\ u : (\tau', (P1, SP_2), \Gamma_1)}\ \texttt{(APP1)}$$

$$\frac{\begin{array}{l} P;\,\Gamma \vdash t : (\tau_1, P_1, \Gamma_1) \\ S' = \mathit{unify}(\tau_1 = a \to b) \quad a,b\ \textit{fresh} \\ P;\,\Gamma_1 \vdash^{\wedge_I} [S'a]\ u : (P_2, \Gamma_2, S) \end{array}}{P;\,\Gamma \vdash t\ u : (SS'b, (SP1, P_2), \Gamma_2)}\ \texttt{(APP2)}$$

$$\frac{\begin{array}{l} \emptyset;\,\Gamma \vdash^{gen} u : (\sigma, \Gamma'') \\ P;\,\Gamma''_x, (x:\sigma) \vdash t : (\tau, P', \Gamma') \end{array}}{P;\,\Gamma \vdash \texttt{let}\ \ x = u\ \texttt{in}\ \ t : (\tau, P', \Gamma'_{[x,\Gamma]})}\ \texttt{(LET)}$$

$$\frac{\begin{array}{l} P;\,\Gamma \vdash^{gen} t : (\sigma', \emptyset, \Gamma') \\ \vdash^{sh} \sigma' \leq \sigma \\ \vdash^{inst} \sigma \leq P' \Rightarrow \tau \end{array}}{P;\,\Gamma \vdash (t :: \sigma) : (\tau, P', \Gamma')}\ \texttt{(ANNOT)}$$

$$\boxed{P;\Gamma \vdash^{gen} t : (\sigma, \Gamma')} \qquad \boxed{\vdash^{inst} \sigma \leq \rho}$$

$$\frac{\begin{array}{l} P;\,\Gamma \vdash t : (\tau, Q, \Gamma') \\ ftv(Q - Q\mid^*_{ftv(\tau)}) = \emptyset \\ \overline{a} = ftv(Q \Rightarrow \tau) - ftv(\Gamma') \end{array}}{P;\Gamma \vdash^{gen} t : (\forall \overline{a}.\, Q \Rightarrow \tau, \Gamma')}\ \texttt{(GEN)}$$

$$\frac{\begin{array}{l} Q \models [\overline{a \mapsto b}]P \\ \overline{b}\ \textit{fresh} \end{array}}{\vdash^{inst} \forall \overline{a}.\, P \Rightarrow \tau \leq Q \Rightarrow [\overline{a \mapsto b}]\tau}\ \texttt{(INST)}$$

$$\boxed{\vdash^{sh} \sigma \leq \sigma'}$$

$$\frac{\begin{array}{l} S = \mathit{match}(\tau \rhd \tau') \\ Q \models SP \qquad \overline{b}\ \textit{fresh} \end{array}}{\vdash^{sh} \forall \overline{a}.\, P \Rightarrow \tau \leq \forall \overline{b}.\, Q \Rightarrow \tau'}\ \texttt{(SH)}$$

Figura 4.3. Inferência de tipos para `CTi`

parâmetro de tipo interseção no corpo da função, que corresponde ao caso em que é possível mais de uma escolha dentre os tipos que ocorrem na interseção. Essa situação é tratada por meio do julgamento de inferência de tipo auxiliar $P;\,\Gamma_2 \vdash^{\wedge_E} [\tau]x : (P_1, \Gamma_1, S, \tau')$.

- A regra `(APP2)` corresponde ao caso em que o termo t não é uma variável. Nesse caso, deve ser considerada a possibilidade de que o tipo de t seja da forma $\wedge\overline{\tau} \to \tau'$, o que requer "verificar" se o tipo do argumento pode ser "promovido" para o tipo $\wedge\overline{\tau}$. Portanto, nessa forma da regra da aplicação, o tipo da função deve ser inferido primeiro, sendo usado para guiar a inferência do tipo do argumento. Essa situação é tratada por meio do julgamento auxiliar de inferência de tipos $P;\Gamma \vdash^{\wedge_I} [\iota]t : (P', \Gamma', S)$.

Note que a escolha dentre as duas possíveis regras de inferência de tipo para uma aplicação é determinada pela forma sintática do termo que corresponde à função.

O julgamento auxiliar de inferência de tipo

$$P;\Gamma \vdash^{\wedge_I} [\iota]t : (P', \Gamma', S)$$

significa que se pode "verificar" que o tipo inferido para t no contexto $P;\Gamma$, digamos τ', pode ser usado em um contexto que "requer" um valor de tipo τ, isto é, τ' unifica com τ, ou pode ser "promovido" para o tipo τ, quando τ é um tipo interseção da forma $\wedge\overline{\tau}$. As regras para derivação desse julgamento são apresentadas na Figura 4.4, sendo que a regra a ser usada é determinada pela forma sintática do tipo ρ.

O julgamento da forma

$$P;\ \Gamma \vdash^{\wedge_E} [\tau]\ x : (P, \Gamma, S, \tau')$$

corresponde aos dois possíveis casos da regra de instanciação do sistema de tipos. Nesse julgamento, τ é o tipo inferido para o parâmetro ao qual x deve ser aplicado, sendo (τ', P, Γ) a tipagem inferida para o resultado dessa aplicação, em um contexto $P;\,\Gamma$. As regras para derivação desse julgamento são apresentadas na Figura 4.5.

Note que a regra `(INSTi)` usa *match*, e não unificação: o objetivo é selecionar, dentre os tipos componentes do tipo interseção, aquele que pode ser usado como tipo da função em uma aplicação a um argumento de tipo (τ). A função *match* pode ser definida em termos da unificação:

$$match(\tau_1 \rhd \tau_2) = unify(\tau_1 = [\overline{a \mapsto c_K}]\tau_2)$$
onde $\overline{a} = ftv(\tau_2)$ e $\overline{c_K}$ são novas constantes de Skolem

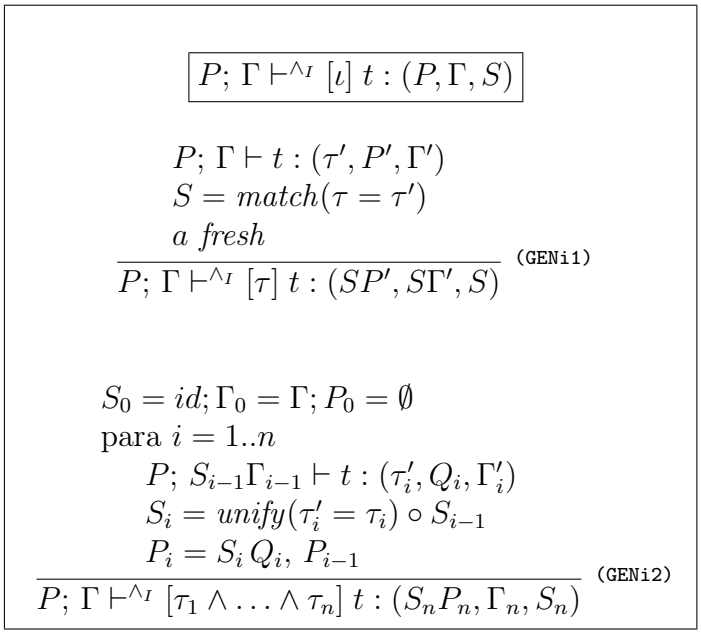

Figura 4.4. Inferência de tipo da aplicação, dirigida pelo tipo da função

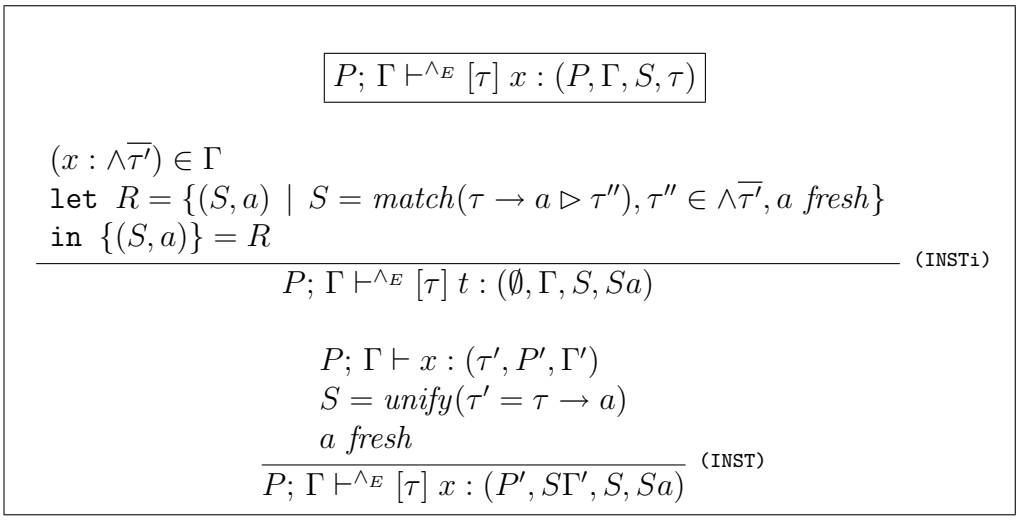

Figura 4.5. Inferência de tipo da aplicação, dirigida pelo tipo do argumento

Um protótipo desse algoritmo de inferência de tipos foi implementado na linguagem Haskell, para um subconjunto de expressões nessa linguagem. Essa implementação é baseada na implementação do sistema de sobrecarga de Haskell descrito em [23] e é descrita na seção a seguir.

Embora não tenhamos ainda uma prova formal das propriedades de correção e de tipo principal para o algoritmo de inferência de tipos proposto acima, temos forte convicção da validade dessas propriedades, exceto para o caso de anotações de tipo

interseção em parâmetros de função, exceto no caso em que tal anotação é tal que um dos usos do parâmetro no corpo da função corresponde a mais de um dos tipos que compõem o tipo anotado. Esse caso é tratado adequadamente na inferência de tipos, não sendo permitido, já que corresponde a uma ambiguidade, mas é permitido no sistema de tipos.

A similaridade entre os sistemas de tipo `CTi` e `CTH`, e os testes realizados com o protótipo implementado, nos permitem formular as seguintes conjecturas em relação às propriedades de correção e inferência de tipos do algoritmo apresentado neste trabalho (excetuando-se o caso de expressões com anotação de tipo tal como mencionado acima).

Conjectura 4.4.1 (Correção) *Para todo termo t, se $P;\Gamma \vdash t : (\rho, P', \Gamma')$ então $P';\Gamma' \vdash t : \rho$ (exceto se t é uma função com anotação de tipo interseção em que um dos usos do parâmetro no corpo da função corresponde a mais de um tipo incluído no tipo interseção anotado, caso em que existe a derivação de tipo no sistema, mas o algoritmo detecta, corretamente, erro de tipo).*

Conjectura 4.4.2 (Tipo Principal) *Para todo termo t, se $P;\Gamma \vdash^{gen} t : \sigma$, então $P;\Gamma \vdash^{gen} t : (\rho', \Gamma')$, onde $\vdash^{sh} \Gamma' \leq \Gamma$, e $\vdash^{sh} \sigma' \leq \sigma$.*

4.5 Implementação

A implementação da inferência de tipos para o sistema `CTi` foi baseada em uma implementação do sistema descrito em [4] e na implementação descrita por Mark P. Jones em [23]. Esse código está que está disponível em `https://github.com/rodrigogribeiro/core/tree/typeinference` e o código da implementação da inferência de tipos para `CTi` aqui descrita está disponível em `https://github.com/emcardoso/CTi`.

A inferência de tipos é implementada sobre uma linguagem núcleo, para a qual é traduzido um programa escrito na linguagem Haskell original. A representação de tipos nessa linguagem núcleo foi modificada, de modo a incluir também uma representação apropriada para tipos interseção. Tipos interseção são representados como uma lista de tipos. Como vimos, no sistema `CTi` tipos interseção apenas podem ser introduzidos por meio de anotações de tipos, e apenas podem ocorrer em parâmetros de funções. A informação provida pela anotação de tipo deve ser usada para guiar o processo de inferência de tipos. Em virtude disso, é necessário modificar o método tradicional de inferência de tipos, de modo a propagar a informação provida pela anotação de tipo ao longo do código do programa, usando usa essa informação quando necessário.

Considere o seguinte exemplo:

```
f :: (Int -> Int) -> (Int,Int)
f g = (g 1, g 2)
```

Na inferência de tipos tradicional, o tipo (do corpo) de `f` é inferido e verifica-se se o tipo anotado é uma instância do tipo inferido, indicando erro em caso contrário. Note que, nesse caso, o tipo inferido para o parâmetro `g` seria $Num\ a \Rightarrow a \rightarrow b$, sendo o tipo especificado para o parâmetro, na anotação do tipo de `f`, uma instância do tipo inferido.

Considere agora o seguinte exemplo:

```
f :: ( Char -> Char ^ Bool -> Bool) -> (Char,Bool)
f g = (g 'x', g False)
```

Neste caso, seria indicado erro de tipo na ausência da anotação de tipo, devido ao fato de que o parâmetro `g` é aplicado a argumentos de tipos distintos no corpo de `f`. Entretanto, a anotação de tipo acima provê uma informação que permite eliminar a restrição do sistema `HM` que exige parâmetros tenham tipo monomórfico – a anotação de tipo informa explicitamente que `g` deve poder ser usado no corpo de `f` tanto com tipo `Char -> Char`, como com tipo `Bool -> Bool`. Portanto o inferidor de tipos deve propagar essa informação, de maneira a inferir o tipo do corpo de `f` em um contexto de tipos em que seja atribuída a `g` a lista de tipos `[Char -> Char, Bool -> Bool]`.

Para entender como é feita propagação dessa informação, precisamos explicar brevemente os tipos de dados usados para representar expressões e tipos na linguagem núcleo. Programas são representados como coleções de *binding groups*, que consistem de associações entre nomes de símbolos e expressões, podendo ou não incluir uma anotação de tipo. Nosso caso de interesse é um *binding group* que inclua anotações de tipo.

```
> data Bind a = FunBind a (Maybe (Scheme a)) (Maybe (Alts a))
>             | PatBind (Pat a) (Maybe (Scheme a)) (Expr a)
>               deriving (Eq, Ord, Show, Data, Typeable)
>
> type Binds a = [Bind a]
```

O tipo de dado algébrico `Bind a` representa um *binding group*. O construtor de tipo `FunBind` é usado para associar um nome a um conjunto de alternativas de definições de expressões, cada uma associada a um respectivo padrão. Um padrão é tal como um padrão na linguagem Haskell — por exemplo, um padrão de lista vazia (`[]`), um padrão de tupla `(x,y)`, e assim por diante. Alternativas são representadas

como pares `([Pat a], Expr a)`. Um `Scheme` é um tipo polimórifico, que pode ou não incluir restrições (de classe). Tipos são representados pelo tipo de dados `Ty a` a seguir:

```
> data Ty a = TVar (TyVar a)
>           | TCon (TyCon a)
>           | TyApp (Ty a) (Ty a)
>           | TyFun (Ty a) (Ty a)
>           | TyTuple [Ty a]         -- Unit type is TyTuple []
>           | TyList (Ty a)
>           | TyAnd [Ty a]           -- for Insersection Type annotation.
>           deriving(Eq, Ord, Show, Data, Typeable)
```

Os construtores de tipos `TyApp` e `TyFun`, apesar de sintaticamente semelhantes, tem semânticas diferentes, uma vez que o primeiro representa o tipo da aplicação de um contrutor de tipos a um outro tipo, e o segunrdo representa o tipo de funções. Ao conjunto de construtores de tipo original foi adicionado o construtor `TyAnd`, para a representação de tipos interseção.

A sintaxe de expressões é representada pelo tipo de dado `Expr` a seguir, cujos construtores são autoexplicativos.

```
> data Expr a = Var a
>             | Con a
>             | Const (Lit a)
>             | Lam [Pat a] (Expr a)
>             | App (Expr a) (Expr a)
>             | Case (Expr a) (Alts a)
>             | If (Expr a) (Expr a) (Expr a)
>             | List [Expr a]
>             | Tuple [Expr a]
>             | Let (Binds a) (Expr a)
>             deriving (Eq, Ord, Show, Data, Typeable)
```

As informações dadas pela anotação do tipo de uma função são propagadas para a inferência do tipo do corpo dessa função, na forma de um contexto de tipos que contém informação do tipo de cada parâmetro. Na nossa implementação, na inferência dos tipos de definições em um *binding group* (`FunBind`), o contexto obtido das anotações de tipo contidas no *binding group* é passado para a função `tcExpl`, que infere o tipo de cada uma das definições alternativas (`tcAlts'`). O trecho de código de `tcExpl` que corresponde à inferência de tipos para um *binding group* é apresentado a seguir.

```
> tcExpl :: Bind Name ->  TcM [Pred Name]
> tcExpl (FunBind n (Just sc) (Just alts))
>           = do
>                (ps :=> t) <- freshInst sc
>                (ps', t') <- tcAlts' (skol t) alts
>                s' <- match t' t
>                s <- wf (apply s' $ ps' :=> t')
>                ps'' <- improve ( apply (s @@ s') ps')
>                pc   <- improve ( apply (s @@ s') ps )
>                sc'' <- quantify (apply (s @@ s') (ps'' :=> t'))
>                sc'  <- quantify (pc :=> apply (s @@ s') t)
>                if sc' /= sc'' then
>                    throwError ("Type Signature error in\n" ++
>                                (show $ pUnlines [(n :>: sc), (n :>: sc')]))
>                  else return (apply (s @@ s') ps
>                ..............
```

A função `skol`, usada no corpo de `tcExpl`, simplesmente transforma em constantes de tipo as variáveis de tipo que ocorrem em tipos interseção. Isto é feito para evitar que estas variáveis sejam inadequadamente instanciadas, situação que poderia ocorrer se o tipo inferido para a variável fosse mais específico que o tipo anotado pelo programador, o que é indicado como um erro de anotação de tipo.

A função `tcAlts'` simplesmente chama a função `tcAlt'`, a qual infere o tipo de cada alternativa individualmente. O código dessas funções é apresentado a seguir. Note que o tipo passado para `tcAlts'` é repassado para `tcAlt'`. A função `args`, definida em `tcAlt'`, associa cada padrão ao tipo correspondente na assinatura de tipos provida pelo programador, gerando uma lista de pares `(Pat Name,Ty Name)`, onde `name` representa um nome de símbolo, e `Ty Name` é um tipo. A função `tcPats` usa essa lista de associações e constrói um novo contexto de tipos, `as`, que por sua vez é usado na verificação do tipo da expressão associada ao padrão.

```
> tcAlts' :: Ty Name -> Alts Name -> TcM ([Pred Name], Ty Name)
> tcAlts'  t alts = do
>                       psts <- mapM (tcAlt' t ) alts
>                       let
>                           ps'     = concatMap fst psts
>                           (t:ts)  = map snd psts
>                       mapM_ (unify t) ts
```

```
>                        s <- getSubst
>                        return (apply s ps', apply s t)
>
> tcAlt' :: Ty Name -> Alt Name -> TcM ([Pred Name], Ty Name)
> tcAlt' t (pats, e) = do
>                      (ps, as, ts) <- tcPats' (args pats t)
>                      (qs, t')      <- context as (tcExpr e)
>                      return (ps ++ qs, foldr TyFun t' ts)
>    where
>      args [] t = []
>      args (pat:pats) (TyFun arg res) = (pat,arg) : args pats res
>      args (pat:pats) _               = error " Invalid type annotation."
```

Durante a inferência de tipos de expressões, basicamente duas situações de interesse podem ocorrer: 1) a aplicação de um valor de tipo interseção a um valor cujo tipo não é um tipo interseção; 2) a aplicação de uma função, cujo parâmetro tem tipo interseção, a um valor de tipo polimórfico. Para tratar essas situações é necessário modificar, na implementação original, apenas o caso de inferência de tipos para a aplicação.

O exemplo apresentado anteriormente ilustra o caso da situação 1): o parâmetro `g` tem tipo interseção e é aplicado a valores de tipos distintos, no corpo da função `f`. Nesse caso, o tipo de `g` deve ser especializado para o tipo do argumento, em cada aplicação. No caso da aplicação `g 'x'` o tipo de `g` deve ser especializado para `Char -> Char`. Essa especialização é realizada por meio de *matching* do tipo $T \rightarrow a$, onde T é o tipo do argumento de `g` e a é uma variável de tipo *fresh*, com cada um dos tipos presentes no tipo interseção anotado para `g`. Dentre esses possíveis *matching*, exatamente um deve ser bem sucedido. Se nenhum *matching* for bem sucedido, o tipo interseção de `g` não atende o que é requerido pelo argumento da aplicação. Se mais de um *matching* for bem sucedido, temos uma situação de ambiguidade, em que não seria possível determinar qual o tipo e, correspondentemente, qual q implementação, de `g` deveria ser usada nessa aplicação. Esses dois casos são considerados erro de tipo.

Considere agora uma aplicação tal como `f id`, onde `f` é a função definida acima, e `id` é a função identidade, cujo tipo é inferido como `id ::` $\forall a.\, a \rightarrow a$. Essa aplicação corresponde ao caso 2) mencionado: o tipo do argumento de `f`, ou seja, o tipo de `id`, deve poder ser "generalizado" para o tipo interseção do parâmetro de `f` para que a aplicação seja bem tipada. Mais especificamente, o tipo de `id` deve ser "generalizado" para o tipo interseção representado pela lista de tipos `[Char -> Char, Bool`

`-> Bool]`. Isso é feito do seguinte modo:

1 Obtém-se um tipo τ' tal que τ' é obtido por *fresh isntantiation* do tipo do argumento. Nesse caso, teríamos $\tau' = a_1 \rightarrow a_1$, onde a_1 é uma variável *fresh*.

2 Para cada tipo τ pertencente ao tipo interseção `[Char -> Char, Bool -> Bool]`, faz-se *matching*(τ,τ').

3 O tipo τ para o qual o *matching* é bem sucedido é eliminado da lista de tipos (interseção).

4 O processo é repetido, a partir do passo 1, até que não reste nenhum tipo na lista (interseção).

5 Para que a aplicação seja bem tipada, *todos* os *matching* devem ser bem sucedidos.

Por fim, para cada aplicação é necessário determinar se corresponde ao caso 1) especialização do tipo da função, ou ao caso 2) "generalização" do tipo do argumento para um tipo interseção. Para isso, em uma aplicação fx, primeiro é inferido o tipo de f, para verificar se esse tipo tem parâmetro de tipo interseção; se este for o caso, o tipo de x deve ser generalizado; caso contrário, o tipo de `f` pode ter que ser especializado (se for um tipo interseção).

4.6 Conclusão

Neste capítulo foi definido o sistema `CTi` e apresentado um algoritmo para inferência de tipos nesse sistema. O sistema `CTi` constitui uma extensão conservativa do sistema (Hindley-Milner + suporte a sobrecarga por meio de classes de tipos), com a introdução de uma forma restrita de polimorfismo de ordem superior, por meio da possibilidade de especificar o tipo de um parâmentro de função como um tipo interseção.

Um protótipo do algoritmo de inferência de tipos apresentado neste capítulo foi implementado na linguagem Haskell, para um subconjunto de expressões dessa linguagem.

A forma de polimorfismo de rank superior suportada pelo sistema `CTi` é complementar ao recurso de polimorfismo paramétrico de rank superior. Existem expressões que podem ser tipadas no sistema `CTi`, tais como as diversas aplicações da função `foo` discutidas anteriormente, mas que não podem ser tipadas em sistemas de polimorfismo de rank superior, se for provida uma única definição para a função `foo`. Em um sistema

de polimorfismo de rank superior, somente seria possível tipar essas diversas aplicações se fossem providas várias definições para `foo`, com anotações de tipo distintas, apropriadas para o uso de `foo` em cada uma dessas aplicações, embora em todas a função `foo` fosse definida pela mesma expressão.

Por outro lado, em um sistema com suporte para polimorfismo paramétrico de rank arbitrário, seria possível atribuir tipo a uma expressão tal como `map f xs`, onde `f` é uma função polimórfica, de tipo, digamos, `f` $:: \forall a, b.\, ([a] \to [a]) \to b$, e `xs` é uma lista de elementos polimórficos, de tipo `xs` $:: [\forall a.\, ([a] \to [a])]$. Essa aplicação é possível em um sistema em que o tipo de `map` possa ser instanciado impredicativamente. Uma anotação de um tipo de rank superior para uma função pode também ser usada para "restringir" o comportamento da mesma, garantindo determinadas propriedades, que podem ser derivadas a partir do seu tipo. Isso é útil para a prova de propriedades de programas, como é mostrado em [50].

Capítulo 5

Conclusão

Neste trabalho apresentamos a definição de um sistema de tipos que constitui uma extensão conservativa do sistema (Hindley Milner + suporte a sobrecarga por meio de classes de tipos), provendo como recurso adicional uma forma restrita de polimorfismo de ordem superior, por meio da possibilidade de especificar o tipo de um parâmetro de função como um tipo interseção.

Esse recurso permite definir funções cujo parâmetro pode ser usado com diferentes tipos no corpo da definição dessa função, podendo tal função ser aplicada tanto a argumentos que são funções de tipo polimórfico paramétrico como funções sobrecarregadas.

Embora existam diversos trabalhos sobre sistemas de tipos para suporte a sobrecarga, assim como inúmeros trabalhos sobre sistemas de tipos interseção, a combinação desses recursos em um mesmo sistema de tipos, tal como apresentada neste trabalho, é completamente original.

Como discutimos no final do Capítulo 4, a forma de polimorfismo de rank superior suportada pelo sistema `CTi`, definido neste trabalho, é complementar ao recurso de polimorfismo paramétrico de rank superior, existindo expressões que podem ser tipadas em um sistema, mas não no outro, e vice versa.

O algoritmo para inferência de tipos para expressões do sistema `CTi` descrito neste trabalho foi implementado em Haskell, para um subconjunto de expressões dessa linguagem, naturalmente estendido com a possibilidade de serem especificadas anotações de tipos em que tipos interseção ocorrem do lado esquerdo do construtor de tipos funcionais.

5.1 Trabalhos Futuros

Alguns aspectos desse trabalho devem ser ainda concluídos ou merecem ser considerados, sendo planejados como trabalhos futuros:

1. A prova das propriedades de correção e de tipo principal para o algoritmo de inferência de tipo para o sistema `CTi`.

2. Uma descrição detalhada da implementação do algoritmo de inferência de tipos.

3. Uma definição formal da semântica de expressões desse sistema e a prova da propriedade de correção so sistema de tipos em relação a essa semântica.

4. A definição de uma versão desse sistema de tipos na forma de um "sistema de tipos bidirecional", com regras de derivação e verificação de tipos, e a definição do algoritmo de inferência de tipos correspondente, com o objetivo de diminuir o requerimento de anotações de tipo em linguagens baseadas no sistema `CTi`, por meio da propagação dessas anotações de tipo ao longo do processo de inferência de tipos.

5. A implementação de um protótipo desse novo algoritmo de inferência de tipos.

Uma extensão interessante do sistema de tipos aqui proposto seria a sua utilização como base para a definição de uma sistema ainda mais flexível, que possibilite a passagem de funções sobrecarregadas g como argumento para uma função f, de maneira que, no corpo de f, g possa ser aplicada a estruturas de dados com elementos de tipos heterogêneos, tais como uma lista heterogênea. A idéia é definir o tipo dessa lista heterogênea como um tipo $[T]$, em que t representa todos os possíveis tipos de elementos dessa lista; uma lista com tal tipo poderia ser usada seguramente (sem possibilidade de erros de tipo) como argumento de uma função sobrecarregada g, se, para cada um dos tipos representados por T, existe uma definição de uma instância em que o parâmetro de g tem esse tipo.

Referências Bibliográficas

[1] Henk Barendregt. *Lambda Calculus: ist Syntax and Semantics*. North Holland, 1984.

[2] Henk Barendregt, S. Abramsky, D. M. Gabbay, T. S. E. Maibaum, and H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[3] Carlos Camarão and Lucília Figueiredo. Type inference for overloading, 1999.

[4] Carlos Camarão, Rodrigo Ribeiro, Lucília Figueiredo, and Cristiano Vasconcellos. A solution to haskell's multi-paramemeter type class dilemma. In *Proceedings of 13th Brazilian Symposium on Programming Languages*, pages 19–21, 2009.

[5] Carlos Camarão, Cristiano Vasconcellos, Lucilia Figueiredo, and João Nicola. Open and closed worlds for overloading: a definition and support for coexistence. *Journal of Universal Computing*, 13(6):874–890, 2007.

[6] Manuel Chakravarty, Gabrielle Keller, and Simon Peyton Jones. Associated type synonyms. In *ICFP'05: ACM International Conference on Functional Programming*, 2005.

[7] D. Clement, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ml. In *ACM Symposium on Lisp and Functional Programming*, pages 13—-27. ACM Press, 1986.

[8] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame J. Formal Logic*, 21(4):685–693, 1980.

[9] M Coppo, M Dezani-Ciancaglini, and B. Veneri. *Principal type schemes and lambda-calculus semantics*. Academic Press, 1980.

[10] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL '82: Symposium on Principles of Programming Languages*, pages 207–212. ACM, 1982.

[11] Atze Dijkstra. *Stepping through Haskell.* PhD thesis, Utrecht University, 2005.

[12] Gregory J. Duck, Simon Peyton-Jones, Peter J. Stuckey, and Martin Sulzmann. Sound and decidable type inference for functional dependencies, 2004.

[13] Dominic Duggan and John Ophel. Type-checking multi-parameter type classes. *Journal of Functional Programming*, 12(2):133–158, 2002.

[14] Jean-Yves Girard. Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur. thèse d'etat. In J. E. Fenstad., editor, *Proceedings of the Second Scandinavian Logic Symposium*, 1972.

[15] Cordelia Hall, Kevin Hammond, Simon Peyton-Jones, and Philip Wadler. Type classes in haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):109–138, 1996.

[16] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146, 1969.

[17] P. Hudak, R. J. M. Hughes, S. Peyton-Jones, and P. Wadler. A history of haskell: being lazy with class. In *History of Programming Languages*, pages 12–1–12–55. ACM Press, 2007.

[18] Paul Taylor Jean-Yves Girard, Yves Lafont. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science, 1989.

[19] Mark P. Jones. A theory of qualified types. In Bernd K. Bruckner, editor, *ESOP '92, 4th European Symposium on Programming*, volume 582, pages 287–306. Springer-Verlag, 1992.

[20] Mark P. Jones. Coherence for qualified types. Technical Report YALEU/DCS/RR-989, Yale University, Department of Computer Science, September 1993.

[21] Mark P. Jones. *Qualified Types: theory and practice.* PhD thesis, University of Nottingham, Department of Computer Science, 1994.

[22] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5:1–35, 1995.

[23] Mark P. Jones. Typing haskell in haskell. In *ACM Haskell Workshop*. ACM Press, 1999.

[24] Mark P. Jones. Type classes with functional dependencies. In *ESOP'00: European Symposium on Programming*, volume 1782. LNCS, Springer-Verlag, 2000.

[25] Simon Peyton Jones, Mark P. Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, 1997.

[26] A. J. Kfoury and J. Tiuryn. Type reconstruction in finite fragments of second order lambda calculus. *Information and Computation*, 98(2):228–257, 1992.

[27] A. J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *POPL'99: ACM Conference on Principles of Programming Languages*, pages 161–174. ACM Press, 1999.

[28] Didier Le Botlan and Didier Remy. Mlf: Raising ml to the power of system f. In *ICFP'2003: International Conference on Functional Programming*, pages 27–38, 2003.

[29] Didier Le Botlan and Didier Remy. Recasting mlf. *Information and Computaion*, 207(6):726–785, 2009.

[30] Daan Leijen. Flexible types: robust type inference for first-class polymorphism. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 66–77. ACM Press, 2009.

[31] Dann Leijen. Hmf: Simple type inference for first-class polymorphism. In *ICFP'2008: International Conference on Fucntional Programming*, pages 283–294. ACM Press, 2008.

[32] Dann Leijen. Flexible types: robust type inference for first-class polymorphism. In *POPL'2009: International Conference on Principles of Programming Languages*, pages 66–77. ACM Press, 2009.

[33] Dann Leijen and Andres Löh. Qualified types for mlf. In *ICFP'2005: International Conference on Fucntional Programming*, pages 144–155, 2005.

[34] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML, revised edition.* MIT Press, 1997.

[35] John Mitchell. *Foundations for Porgramming Languages.* MIT Press, 1996.

[36] John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76:211–249, 1998.

[37] M. Odersky and K. Läufer. Putting type annotations to work. In *ACM Symposium on Principles of Programming Languages*, pages 54–67. ACM Press, 1996.

[38] S. Peyton-Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1):1–82, 2007.

[39] Simon Peyton-Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 4 edition, 2003.

[40] Benjamin C. Pierce. *Types and programming languages*. The MIT Press, 1 edition, 2002.

[41] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

[42] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423, London, UK, 1974. Springer-Verlag.

[43] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423. Springer-Verlag, 1974.

[44] J.A. Robinson. Computational logic: The unification computation. *In Machine Intelligence*, 6:63–72, 1971.

[45] C. Shan. Sexy types in action. *ACM SIGPLAN Notices*, 39(5):15—-22, 2004.

[46] C. Strachey. Fundamental concepts in programming languages. *Journal Higher-Order and Symbolic Computation*, 13:11–49, April 2000.

[47] P. J. Stuckey and M. Sulzmann. A theory of overloading. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(27):1—54, 2005.

[48] M. Sulzmann, T. Schrijvers, and P. J. Stuckey. Principal type inference for ghc-style multi-parameter type classes. In *APLAS'06*, volume 4279, pages 26–43. LNCS, Springer-verlag, 2006.

[49] Martin Sulzmann, Gregory J. Duck, Simon P. Jones, and Peter J. Stuckey. Understanding functional dependencies via constraint handling rules. *Journal of Functional Programming*, 17(1):83–129, 2007.

[50] Janis Voitlander. *Types for Programming Reasoning.* PhD thesis, Technische Universität Dresden, 2009.

[51] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy types: Inference for higherrank types and impredicativity. In *ICFP '06: Proceeding of the 11th ACM SIGPLAN International Conference on Functional Programming*, pages 251–262. ACM Press, 2006.

[52] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Fph: first-class polymorphism for haskell. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN International Conference on Functional programming*, pages 295–306. ACM Press, 2008.

[53] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL'89: Symposium on Principles of Programming Languages*, pages 60–76. ACM Press, 1989.

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

[55] J. B. Wells. Typability and type checking in the second-order lambda-calculus are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.