Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computacão

Inferência de Tipos com Suporte para Sobrecarga Baseada no Sistema CT

Cristiano Damiani Vasconcellos

Belo Horizonte
Outubro de 2004

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computacão

Inferência de Tipos com Suporte para Sobrecarga Baseada no Sistema CT

Cristiano Damiani Vasconcellos

Tese apresentada ao Programa de Pós–Graduação em Ciência da Computação da Universidade Federal de Minas Gerais como requisitos parcial para obtenção do título de Doutor em Ciência da Computação.

Orientador: Prof. Carlos Camarão
Co–Orientadora: Profa. Lucília Figueiredo

Belo Horizonte
Outubro de 2004

Cristiano Damiani Vasconcellos

Inferência de Tipos com Suporte para Sobrecarga Baseada no Sistema CT

Tese apresentada ao Programa de Pós–Graduação em Ciência da Computação da Universidade Federal de Minas Gerais como requisito parcial para obtenção do título de Doutor em Ciência da Computação. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Carlos Camarão
Orientador

Profa. Lucília Figueiredo
Co–Orientadora

Prof. Andre Luís de Medeiros Santos
CIn-UFPE

Profa. Mariza Andrade da Silva Bigonha
DCC-UFMG

Prof. Roberto Ierusalimschy
DI-PUC-Rio

Prof. Roberto da Silva Bigonha
DCC-UFMG

Belo Horizonte, 1 de Outubro de 2004

Agradecimentos

Muitas pessoas foram importantes para que eu começasse e concluísse este trabalho, gostaria de agradecer pelo menos algumas delas. Alguns me acompanharam por estes anos de curso, como a minha esposa Aletéia que foi uma grande companheira, me incentivando e principalmente me aturando nas piores fases. Meu orientador Carlos Camarão que desde o começo acreditou em mim e com muita paciência, em nossos longos encontros à tarde, conseguiu abrir minha cabeça, aguentando no início minha inexperiência e no final minha teimosia, muitas vezes infundada. A minha co-orientadora Lucília Figueiredo por ter me aceito tão bem no grupo e principalmente pelo apoio nesses últimos dias. Outros me influenciaram muito na escolha dos caminhos que me levaram a chegar até aqui, como o meu ex-professor e agora amigo Cláudio Oliveira que me fez dar os primeiros e os vários outros passos que se seguiram no estudo de linguagens de programação, e o professor Flávio Bortolozzi com quem sempre pude contar nos momentos difíceis e cujo exemplo me inspirou a seguir a carreira acadêmica.

Gostaria de agradecer também a meus pais que sempre me apoiaram de forma incondicional. E por fim agradeço a Pontifícia Universidade Católica do Paraná, onde fui aluno, estagiário, funcionário e agora professor, que financiou este curso.

Resumo

Este trabalho aborda o problema da inferência de tipos com definições sobrecarregadas, apresentando uma revisão do sistema de tipos CT. Esse sistema é uma extensão do sistema proposto por *Damas-Milner* com suporte para sobrecarga de nomes, onde *restrições de tipo* são usadas em tipos polimórficos para indicar restrições de instanciação desses tipos, de acordo com definições sobrecarregadas existentes. A inferência de tipos nesse sistema envolve a resolução para o problema da satisfazibilidade de restrições, que trata de decidir se um dado conjunto de restrições de um tipo polimórfico é válido ou não, em um determinado contexto de tipos. Políticas para resolução de sobrecarga muito restritivas têm sido adotadas para garantir a decidibilidade deste problema. O sistema CT adota uma política de resolução de sobrecarga muito menos restritiva que as presentes em sistemas de tipos similares. As principais contribuições desse trabalho são: uma definição formal do problema de satisfazibilidade de restrições independente das regras de derivação do sistema de tipos, apresentação de um algoritmo para a solução desse problema e a implementação de um protótipo que demonstra que a adoção de uma política de sobrecarga menos restritiva pode funcionar bem na prática. Nos testes realizados com a implementação desse algoritmo, usando código similar a programas implementados em *Haskell*, não foi encontrada nenhuma situação em que a satisfazibilidade das restrições não pôde ser decidida. Para estes casos é utilizado um limite de iteração para interromper o processo e rejeitar a expressão, indicando a ocorrência de um erro de tipo.

Palavras–chave

Linguagens de Programação, Sistemas de Tipos, Inferência de Tipos, Polimorfismo, Sobrecarga.

Abstract

This work discusses the problem of type inference in the presence of overloading, making a revision of type system CT. This system is an extension of the well-known *Damas-Milner* system with support for overloading, where *constraints* are used in polymorphic types to indicate restrictions on the instantiation of these types, with respect to type assumptions of overloaded symbols that are available in the relevant typing context. Type inference in this system involves a solution to the problem of constraint-set satisfiability in a given typing context, that is, the problem of deciding whether a given set of constraints in a polymorphic type is considered valid (is satisfied) or not, in a given typing context. Over-restrictive overloading policies have been adopted in order to guarantee the decidibility of this problem. System CT adopts an overloading policy that is much less restrictive than those of similar type systems. The main contribuitions of this work are the following: a formal definition of the problem of constraint-set satisfiability that is independent of (the rules used in) a type system, the definition of an algorithm for solving this problem and the implementation of a prototype of this algorithm that indicates that the adoption of a less restrictive overloading policy can work well in practice. In tests with this prototype, no situation was detected where the satisfiability could not be decided. In this cases a user configurable iteration limit is used in order to stop the process and reject the expression, indicating a type error.

Keywords

Programming Languages, Type System, Type Inference,Polymorphism, Overload.

Conteúdo

Lista de Figuras

Lista de Tabelas

1 Introdução

O uso de tipos em linguagens de programação tem objetivos diversos, como a detecção de erros, a definição de abstrações, a otimização do código gerado e a documentação de programas. Embora apresentem algum ganho em flexibilidade, linguagens que não adotam verificação de tipos em tempo de compilação (como, por exemplo, *LISP* [38] e *Scheme* [3]) podem tornar o processo de depuração mais demorado, uma vez que erros de tipos são detectados apenas em tempo de execução. A verificação de tipos em tempo de compilação, também chamada de verificação estática, auxilia programadores a escrever códigos corretos, uma vez que todos os erros de tipo são detectados durante o processo de compilação. Um tipo é associado a cada expressão (constante, operador, variável e função), permitindo ao compilador verificar a consistência das declarações com os usos dos nomes introduzidos nessas declarações.

Na maioria das linguagens de programação, como, por exemplo, *Pascal* [23], *C* [33] e *Java* [4], os tipos das variáveis e funções são declarados explicitamente em programas. Algumas linguagens, como *SML* [40], por exemplo, tentam combinar a segurança de uma verificação estática com a flexibilidade de declarações sem a anotação de tipos, não exigindo, embora permitam, anotações de tipos em programas. Em *SML*, o tipo de uma expressão é inferido de acordo com o conjunto de nomes que são visíveis no trecho de programa onde a expressão é usada. Tais linguagens adotam ainda sistemas de tipos polimórficos, que permitem que funções (polimórficas) sejam utilizadas com tipos diferentes, determinados de acordo com o contexto em que as funções são usadas. Algumas variações de polimorfismo podem ser encontradas nas linguagens de programação modernas, como descrito sucintamente a seguir.

O *polimorfismo paramétrico* permite definir funções que funcionem de maneira uniforme para tipos distintos, sendo esses tipos, para cada função, instâncias de um único tipo mais geral, comumente chamado simplesmente de tipo da função. Variáveis de tipo são utilizadas nesses tipos, chamados de polimórficos, e são instanciadas para um tipo particular quando necessário. A forma de polimorfismo paramétrico mais conhecida é chamada de *polimorfismo-*

via-let[1], *polimorfismo no estilo ML* ou *polimorfismo de Damas-Milner* [39]. As características principais desse tipo de polimorfismo paramétrico são, em primeiro lugar, que não é permitido que funções tenham parâmetros polimórficos (isto é, parâmetros que requeiram tipos distintos em pontos distintos de seu uso) e, em segundo lugar, não é permitido que uma função seja usada de forma polimórfica na sua própria definição. A eliminação da primeira restrição caracteriza a extensão do polimorfismo de Damas-Milner conhecida como *abstração polimórfica*, e a eliminação da segunda restrição caracteriza a chamada *recursão polimórfica* (ou *polimorfismo de Milner-Mycroft* [42]). Como exemplos de construções baseadas em polimorfismo paramétrico existentes em linguagens de programação podemos citar também *templates* em *C++* [53] e, mais recentemente, *generics* em *Java*.

O *polimorfismo ad-hoc* é baseado em sobrecarga de funções, ou seja, na existência de várias definições que têm um mesmo nome (ou símbolo). A idéia básica é simplesmente que a definição a ser usada deve ser escolhida de acordo com o tipo requerido no ponto do programa em que o nome é usado. Por exemplo, o operador de adição (+) é sobrecarregado em diversas linguagens de programação, para adição de valores de tipo inteiro e para valores de ponto flutuante. Ao contrário do polimorfismo paramétrico, a sobrecarga teve sua importância subestimada até recentemente. Por exemplo, na linguagem *SML* uma política de tratamento de sobrecarga é adotada no caso de operadores aritméticos e outra política é usada no caso da sobrecarga do operador de igualdade. Para operadores aritméticos, como o operador de adição, por exemplo, a sobrecarga deve ser sempre resolvida (isto é, qual definição deve ser usada deve ser sempre determinada) de acordo com o contexto em que o símbolo ocorre. Não é possível, por exemplo, definir o tipo polimórfico de funções como:

$$double = \lambda x.\, x + x$$

No caso do operador de igualdade, é usado um *tipo igualdade*, polimórfico, que usa uma variável de tipo especial, ou seja, com uma notação específica para indicar a necessidade de instanciação para tipos para os quais exista uma definição para o operador de igualdade. Essa variável pode ser instanciada então para todos os tipos básicos e para os tipos criados a partir destes, para os quais a igualdade é verificada comparando os componentes correspondentes (igualdade estrutural). Se por um lado esta solução permite a definição e tipagem correta de funções polimórficas que utilizam o operador de igualdade, por outro lado essa abordagem é bastante limitada, só podendo ser usada para tratar a sobrecarga do operador de igualdade (no máximo, esse esquema poderia ser estendido para

[1]Em inglês *let-polymorphism*

tratar a sobrecarga de um número limitado de operadores predefinidos, para os quais notações especiais teriam que ser escolhidas a priori para denotar os tipos correspondentes).

O *polimorfismo de inclusão* (ou subtipagem), presente nas linguagens orientadas por objetos, caracteriza-se pela seguinte regra: se uma expressão e tem tipo B e B é subtipo de A, então e pode ser tratado como se tivesse tipo A. Isso garante que uma expressão de um dado tipo B possa ser usada em qualquer contexto onde um supertipo de B é esperado [10, 8, 48]. Esse tipo de polimorfismo não faz parte do escopo deste trabalho, estando relacionado no entanto com trabalhos futuros relacionados a sobrecarga de construtores de dados [19].

Muitas das linguagens de programação funcionais modernas adotam o *polimorfismo-via-let*. Esse sistema, originalmente descrito por Milner [39] para a primeira versão de *ML*, foi adotado em *Standard ML*, e posteriormente, possivelmente com algumas variações, em uma série de outras linguagens. O tipo de uma expressão em *ML* é o tipo mais geral possível, também chamado de *tipo principal*, caracterizado pela seguinte propriedade. Um tipo polimórfico é representado na forma $\forall\alpha_1...\alpha_n.\tau$, onde $n \geq 0$ e α_i são variáveis de tipo (sendo, por isso, também conhecidos como tipos quantificados). O tipo principal $\forall\alpha_1...\alpha_n.\tau$ de uma expressão é tal que toda instância desse tipo pode ser obtida pela substituição de zero ou mais variáveis de tipo α_i ($i \in \{1, \ldots, n\}$) por tipos específicos.

A linguagem *Haskell* [16] estendeu o sistema de Damas-Milner adicionando um método flexível e uniforme de tratar sobrecarga — mediante o uso de *classes de tipos* [58, 20] —, que permite sobrecarregar funções para construtores de tipos distintos. Uma declaração de classe introduz uma nova classe de tipos, onde são definidos alguns nomes e seus tipos principais. Esses nomes podem ser então definidos para diversos tipos, em declarações de instâncias da classe. Na declaração de um tipo como instância de uma classe devem ser incluídas definições de cada um dos nomes especificados na declaração da classe. O nome da classe é então utilizado em tipos polimórficos, impondo uma restrição ao conjunto de tipos para os quais uma determinada variável de tipo pode ser instanciada. Essa restrição, representada na forma $\{C\ \alpha\}$, onde C é o nome da classe e α uma variável de tipo, limita os tipos para os quais a variável de tipo α pode ser instanciada aos tipos que são instâncias da classe C. O tipo principal de uma declaração em *Haskell* é da forma $\forall\alpha_1...\alpha_n.\{C_1\ \alpha_1, \ldots, C_m\ \alpha_m\}.\,\tau$, onde $0 \leq m \leq n$ e α_i, $i \in \{1, \ldots, m\}$, é uma variável de tipo que ocorre em τ. Na terminologia de *Haskell*, tipos nos quais ocorrem restrições são chamados de *tipos qualificados*, e uma restrição é também chamada de *predicado*. No predicado $\{C\ \alpha\}$, a variável α é também chamada de parâmetro de C. Em

Haskell 98[2], é permitido especificar apenas um parâmetro para cada classe, embora as implementações mais usadas de interpretadores/compiladores *Haskell* (como *HUGS* e *GHC*) provêem suporte a classes com múltiplos parâmetros, estendendo o mecanismo adotado para classes com um único parâmetro. A previsão é que classes com múltiplos parâmetros sejam incorporadas à linguagem em uma versão futura da linguagem. No entanto, problemas relativos a ocorrência de ambigüidades nos tipos que envolvem símbolos sobrecarregados e classes com múltiplos parâmetros têm atrasado essa adoção[3].

O sistema de tipos *CT*[4] [7] também estende o sistema de tipos proposto por Damas-Milner com suporte a sobrecarga. O tipo de uma expressão em *CT* também contém uma lista de restrições κ, mas agora essa lista é um conjunto de pares $o{:}\tau$, sendo o um símbolo sobrecarregado e τ um tipo simples. Um tipo polimórfico com restrições é escrito na forma: $\forall\alpha_1...\alpha_n.\kappa.\tau$, onde $n \geq 0$. Um *contexto de tipos* Γ é um conjunto finito de *suposições de tipo* $x{:}\sigma$, onde x é um variável ou símbolo e σ um tipo polimórfico com restrições. Um símbolo sobrecarregado contém mais de uma suposição em Γ (a existência de duas ou mais definições para um determinado símbolo em um contexto de tipos indica que este é um símbolo sobrecarregado). A restrição de tipos polimórficos de símbolos (ou nomes) sobrecarregados é definida a partir da generalização mínima (*lcg*[5]) dos tipos de cada uma das definições sobrecarregadas, tornando desnecessárias as declarações de classes.

Ao contrário de *Haskell*, o sistema *CT* baseia-se em uma *abordagem de mundo fechado* para resolução de sobrecarga, que é caracterizada simplesmente por estender o princípio no qual se baseia a tipagem de expressões *sem* o uso de sobrecarga: o tipo de cada expressão é determinado de acordo com o conjunto de declarações visíveis no contexto em que a expressão ocorre. O conjunto de restrições κ que aparece no tipo principal de uma expressão indica que essa expressão pode ocorrer apenas em um contexto onde essas restrições sejam satisfeitas. Caso nesse contexto algum tipo de símbolo sobrecarregado seja instanciado para um tipo tal que não seja possível resolver a sobrecarga, essa ocorrência é rejeitada no processo de verificação da satisfazibilidade de restrições.

De forma diferente das principais linguagens de programação que provêem suporte a sobrecarga (e.g. *Java* e *C++*) o sistema *CT*, a exemplo do sistema de classes de tipos de *Haskell*, adota uma política de resolução de sobrecarga *dependente do contexto*. Uma política independente do contexto simplifica a

[2]Definição oficial da linguagem *Haskell* (a última revisão foi feita em Setembro de 2002).

[3]HUGS e GHC adotam o mecanismo de declaração de dependências funcionais para tratar esse problema, como explicado na Seção 2.2.2.

[4]*Constrained Types*

[5]*least common generalization*

resolução de sobrecarga e a detecção de ambigüidades, mas é restritiva. Por exemplo, símbolos constantes não podem ser sobrecarregados, e não é permitida a sobrecarga de funções onde apenas o valor retornado é diferente para as várias definições. Isso ocorre, por exemplo, no caso de uma função de leitura ou conversão de valores para cadeias de caracteres, como a função *read* definida na biblioteca padrão de *Haskell*. Essa função faz uma análise sintática do conteúdo de uma cadeia de caracteres e retorna o resultado da conversão dessa cadeia para um determinado tipo de dados. A função *read* é sobrecarregada em *Haskell* para os tipos *Int*, *Float*, *Bool*, *String*, entre outros. Cada uma dessas definições tem um tipo que é uma instância do tipo polimórfico $\forall\alpha.\, String \rightarrow \alpha$. Um sistema de tipos que adote uma política dependente do contexto permite a resolução da sobrecarga em declarações como: $\lambda x.\, read\ x$ ==“*abc*”. O tipo de *read*, nesse exemplo, é determinado como sendo $String \rightarrow String$. Se em linguagens como *C++* e *Java* a adoção de uma política independente do contexto é aceitável, em linguagens nas quais funções são valores de primeira ordem, como em *Haskell*, a adoção desta política seria muito restritiva.

Recentemente, têm sido propostas várias extensões do sistema de tipos de Damas-Milner para prover suporte a sobrecarga. Além do sistema de classes de tipos e do sistema CT, podemos citar [31, 58, 11, 52, 20, 44, 14, 50, 54]. Algumas das características desses sistemas são discutidas nas Seções 2.2 e 2.4.

O sistema CT adiciona o tratamento de sobrecarga ao sistema de Damas-Milner impondo um mínimo de restrições e, principalmente mantendo a concepção original de *ML*, onde anotações de tipos não são obrigatórias. Esse trabalho apresenta uma versão revisada do sistema CT, onde a satisfazibilidade de restrições é definida de forma independente do sistema de tipos, sendo as principais contribuições:

- A definição de um algoritmo para verificar a satisfazibilidade das restrições impostas a declarações que usam símbolos sobrecarregados.
- A definição de uma nova regra capaz de identificar expressões ambíguas em uma abordagem de mundo fechado.
- A implementação de um protótipo do algoritmo de inferência de tipos para o sistema CT, a qual possibilita, além da sobrecarga de nomes como definida no sistema CT, também definições polimórficas mutuamente recursivas. A inferência de tipos para definições polimórficas mutuamente recursivas é feita, ao contrário do que ocorre por exemplo em *SML* e *Haskell*, sem que haja necessidade de realizar uma ordenação topológica das declarações para tipagem de expressões. Além disso, a implementação provê suporte a definições com recursão polimórfica, como em *Haskell*, mas

sem requerer anotações de tipo explícitas para definições que apresentam essa característica.

– Um estudo preliminar sobre a viabilidade do uso do sistema *CT* como base para definição de uma linguagem de programação polimórfica, no estilo de *Haskell*, mas sem a necessidade de declarações de classes de tipos. Esse estudo indica partes da implementação cuja eficiência deve ser melhorada, para uso em um compilador/interpretador mais robusto a ser eventualmente usado na prática.

Os demais capítulos estão organizados da seguinte forma. No Capítulo 2 é feita uma breve introdução ao sistema de tipos de Damas-Milner. É descrita, de maneira informal, a extensão desse sistema para suporte a sobrecarga feita em *Haskell*, mediante o uso de classes de tipos. São também discutidas algumas limitações do sistema de classes de tipos, assim como modificações propostas para contornar esses problemas. O Capítulo 3 apresenta uma descrição formal do sistema *CT*, em particular do tratamento dado nesse sistema aos problemas da satisfazibilidade de restrições e de detecção de ambigüidade na resolução de sobrecarga. Nesse capítulo também é discutida a abordagem de mundo fechado do sistema *CT* e uma alternativa para que uma linguagem baseada nesse sistema trabalhe também com a abordagem de mundo aberto. O Capítulo 4 descreve um protótipo de implementação do algoritmo de inferência de tipos do sistema *CT* e apresenta resultados obtidos com a medição de eficiência dessa implementação. O Capítulo 5 conclui o trabalho. A implementação dos algoritmos apresentados para um subconjunto da linguagem *Haskell* pode ser encontrada em:

`http://www.dcc.ufmg.br/~damiani/CT/CT.zip`

2 Fundamentos

Este capítulo aborda conceitos fundamentais para o problema tratado no restante desse trabalho, com enfoque no sistema de tipos de Damas-Milner e no sistema com classes de tipos adotado em *Haskell*.

A descrição formal de um sistema de tipos é normalmente feita por meio da definição de um conjunto de regras para derivação dos tipos possíveis para cada expressão da linguagem. É definido então um algoritmo que, dados uma expressão e um contexto de tipos contendo informações sobre os tipos das variáveis livres da expressão — i.e. variáveis usadas mas não definidas na própria expressão —, determina o tipo principal dessa expressão, que representa todos os tipos que podem ser derivados para a expressão.

É comum dizer que o sistema de tipos "tem a propriedade de tipo principal" se, para qualquer par formado por um contexto de tipos e uma expressão da linguagem, ou não existe tipo derivável para a expressão nesse contexto ou existe um tipo principal para a expressão nesse contexto[13, 17]. Os tipos representados pelo tipo principal são chamados de instâncias desse tipo.

O conceito de tipo principal não deve ser confundido com o conceito similar de *tipagem principal*[24, 48]. Informalmente, temos:

Tipo Principal	
Dado:	um termo e e um contexto de tipos Γ.
Existe:	um tipo σ que representa todos os tipos possíveis de e em Γ.
Tipagem Principal	
Dado:	um termo e.
Existe:	um tipo σ e um contexto Γ tais que Γ contém as suposições de tipo "mínimas" (estritamente necessárias) para tipagem de e, e σ representa todos os tipos que podem ser derivados para e em Γ.

Uma variação simples do conceito de tipagem principal permite fornecer,

como entrada, um contexto de tipos que contém algumas, mas não necessariamente todas, as suposições de tipos necessárias para tipagem de expressões[18]. O conceito de tipagem principal serve como suporte a compilação de módulos separadamente, para compilação incremental e também possibilita a inferência de tipos em expressões polimórficas onde ocorrem definições mutuamente recursivas[24, 18].

2.1 Sistema de Tipos de Damas-Milner

As expressões válidas na mini-linguagem proposta por Robin Milner [39, 13] são definidas pela seguinte sintaxe, onde x representa uma variável, elemento de um conjunto predefinido de variáveis:

$$e \texttt{ ::= } x \texttt{ | } e\ e' \texttt{ | } \lambda x.e \texttt{ | let } x \texttt{ = } e \texttt{ in } e'$$

Nessa mini-linguagem, uma expressão da forma `let` $x = e$ `in` e' pode introduzir uma variável x de tipo polimórfico, de forma que x possa ser usada na expressão e' em contextos que requerem tipos distintos. Pelo fato de variáveis polimórficas serem introduzidas em expressões-`let`, esse tipo de polimorfismo paramêtrico é também conhecido (tipicamente na literatura em língua inglesa) como *polimorfismo-via-let*. Sendo α uma meta-variável de tipo, as expressões de tipos nessa mini-linguagem são dadas pela seguinte sintaxe:

$$\tau \texttt{ ::= } \alpha \texttt{ | } \tau \rightarrow \tau$$
$$\sigma \texttt{ ::= } \tau \texttt{ | } \forall \alpha.\sigma$$

Os tipos são assim divididos em tipos monomórficos (denotados por variáveis τ, τ' etc., possivelmente subscritas) e tipos polimórficos (denotados por σ, σ', etc). Tipos polimórficos são definidos por meio do quantificador universal, sendo por esta razão também chamados de tipos quantificados.

O conjunto de expressões "bem tipadas" é definido pelo sistema de tipos apresentado na Figura 2.1. Fórmulas desse sistema têm a forma $\Gamma \vdash e : \sigma$, significando que a expressão e tem tipo σ no contexto de tipos Γ.

Um contexto de tipos no sistema de Damas-Milner contém apenas uma suposição de tipo para cada variável x. Γ_x representa o contexto Γ mas sem qualquer suposição de tipo para x. A relação $\sigma < \sigma'$ indica que o tipo polimórfico σ é mais geral que o tipo σ'.

$$\Gamma \vdash x : \sigma \qquad (x : \sigma \in \Gamma) \qquad \text{(VAR)}$$

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e' : \sigma'} \qquad (\sigma < \sigma') \qquad \text{(INST)}$$

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall\alpha.\sigma} \qquad (\alpha \text{ não é livre em } \Gamma) \qquad \text{(GEN)}$$

$$\frac{\Gamma \vdash e : \tau \to \tau' \qquad \Gamma \vdash e' : \tau}{\Gamma \vdash (e\ e') : \tau'} \qquad \text{(APPL)}$$

$$\frac{\Gamma_x \cup \{x : \tau'\} \vdash e : \tau}{\Gamma \vdash (\lambda x.e) : \tau' \to \tau} \qquad \text{(ABS)}$$

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma_x \cup \{x : \tau'\} \vdash e' : \tau}{\Gamma \vdash (\text{let } x = e \text{ in } e') : \tau} \qquad \text{(LET)}$$

Figura 2.1: Sistema de tipos de Damas-Milner

Um sistema de tipos "declarativo", como o da Figura 2.1, não provê diretamente um método para inferência de tipos, uma vez que pode existir mais de uma regra a ser usada em determinados casos (ou seja, pode existir mais de uma derivação para uma mesma fórmula $\Gamma \vdash e : \sigma$). Isso ocorre, no caso do sistema da Figura 2.1, devido à existência das regras (INST) e (GEN).

Para inferência de tipos neste sistema, é usado um algoritmo atualmente já bastante conhecido, chamado de *Algoritmo W*. Sua definição se baseia no uso de unificações[1][41].

Uma substituição é uma função de variáveis de tipo em expressões de tipo, estas funções são comumente representadas como $[\tau_1/\alpha_1 ... \tau_n/\alpha_n]$, ou $[\tau_i/\alpha_i]^{i=1..n}$. Substituições são estendidas de forma natural para homomorfismos sobre termos.

Escrevemos simplesmente $S\tau_1 = S\tau_2$ em vez de $S(\tau_1) = S(\tau_2)$ e, em geral, adotamos a convenção (usual) de que a aplicação de substituições é associativa, escrevendo, por exemplo, $SS'S''\alpha$ em vez de $(S \circ (S' \circ S''))(\alpha)$, onde $\circ$ é o operador de composição de funções.

Dois tipos τ_1 e τ_2 são ditos unificáveis se existe uma substituição S tal que $S(\tau_1) = S(\tau_2)$. Nesse caso, a substituição S é chamada de *unificador* dos tipos τ_1 e τ_2. Um unificador S_g é chamado de *unificador mais geral* se, para qualquer outro unificador S, existe uma substituição S' tal que $S' \circ S_g = S$. O *algoritmo W*,

[1]Quando envolve tipos polimórficos o problema de inferência de tipos pode gerar uma instância do problema da unificação de tamanho exponencialmente grande com relação ao tamanho do problema original. A complexidade do problema de inferência de tipos é abordada na Seção 2.3.

$W(\Gamma, x)$ =
  Se $\Gamma(x) = \forall\alpha_1 ... \alpha_2.\tau$ então $(Id, [\beta i/\alpha_i]\tau)$
  senao *Falha*

$W(\Gamma, e\, e')$ =
  let $(S_1, \tau) = W(\Gamma, e)$
      $(S_2, \tau') = W(S_1\Gamma, e')$
      S = *unificar*$(S_2\tau, \tau' \rightarrow \beta)$ onde β é livre
  in $(S \circ S_2 \circ S_1, S\beta)$

$W(\Gamma, \lambda x.e)$ =
  let $(S, \tau) = W(\Gamma_x \cup \{x : \beta\}, e)$
  in $(S, S(\beta \rightarrow \tau))$

$W(\Gamma, let\ x = e\ in\ e')$ =
  let $(S_1, \tau) = W(\Gamma, e)$
      $(S_2, \tau') = W(S_1\Gamma_x \cup \{x : fechamento\ (S_1\Gamma, \tau)\}, e')$
  in $(S_1 \circ S_2, \tau')$

Figura 2.2: Algoritmo W

como apresentado por Damas e Milner [39], tem como entrada um par com um contexto de tipos Γ e uma expressão, e retorna uma substituição e o tipo principal da expressão. Caso a expressão não tenha tipo principal é indicada a ocorrência de um erro. O algoritmo é apresentado na Figura 2.1, sendo que *unificar*(τ_1, τ_2) representa o unificador mais geral para o par de expressões de tipo, e o fechamento de um tipo (quantificação de suas variáveis de tipo) é definido como:

$$fechamento(\Gamma, \tau) = \forall\alpha_1 ... \alpha_n.\tau$$

onde $\alpha_1 ... \alpha_n$ são variáveis de tipo que ocorrem em τ, mas não em Γ.

Robinson [49] apresentou pela primeira vez um algoritmo que obtém o unificador mais geral para dois tipos ou então retorna um erro, caso os tipos não sejam unificáveis.

2.2 Classes de Tipos

Seguindo o exemplo da definição oficial da linguagem *Haskell* [16], é apresentada a seguir uma descrição informal do sistema de tipos dessa linguagem.

Uma definição formal pode ser encontrada em [20] e o algoritmo de inferência em [28].

O sistema de tipos usado na linguagem *Haskell* estende o sistema de tipos de Damas-Milner com *classes de tipos* para permitir a verificação de tipos estática e um tratamento uniforme a sobrecarga. A partir deste ponto, quando nos referimos a *classes* subentenda-se *classes de tipos*. Uma declaração de classe introduz um nome e uma anotação de tipo para cada um dos símbolos a serem sobrecarregados. Uma declaração de instância introduz definições dos símbolos sobrecarregados para um determinado tipo.

Por exemplo, considere a declaração da classe *Eq* a seguir, e a definição de duas instâncias dessa classe:

```
class Eq a where
      (==) :: a → a →Bool
instance Eq Int
      x == y = primEqInt x y
instance Eq Char
      x == y = primEqChar x y
```

Supomos acima que *primEqInt* e *primEqChar* são funções primitivas, com tipos $Int \rightarrow Int \rightarrow Bool$ e $Char \rightarrow Char \rightarrow Bool$, respectivamente, mas essas funções poderiam ser definidas pelo programador. Considerando as declarações acima, em *Haskell* as expressões $2 + 2$ `==` 4 e ´a´ `==` ´b´ são bem tipadas, da mesma forma que declarações polimórficas como:

```
ins a [ ] = [a]
ins a (x:xs) = if a == x then x:xs else x:(ins a xs)
```

Nesse exemplo, o tipo principal inferido para *ins* é: $\forall\alpha.\{Eq\ \alpha\}.\alpha \rightarrow [\alpha] \rightarrow [\alpha]$, sendo $\{Eq\ \alpha\}$ uma restrição sobre o tipo polimórfico $\forall\alpha.\,\alpha \rightarrow [\alpha] \rightarrow [\alpha]$ que limita os tipos para os quais a variável α pode ser instanciada aos tipos que são instâncias da classe *Eq*.

Uma hierarquia de classes pode ser estabelecida com a declaração de classes. Por exemplo:

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

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

A classe *Ord* é definida como subclasse de *Eq*. Dessa forma, um tipo só poderá ser declarado como instância da classe *Ord* se antes ele for declarado como instância da classe *Eq*. O uso de superclasses pode simplificar os tipos de expressões com símbolos sobrecarregados, como no seguinte exemplo:

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

O tipo inferido para *search* é:

$search :: \{Ord\ \alpha\}.\ \alpha \rightarrow [\alpha] \rightarrow Bool$

Sem superclasses teríamos:

$search :: \{Eq\ \alpha,\ Ord\ \alpha\}.\ \alpha \rightarrow [\alpha] \rightarrow Bool$

Ao definir uma classe, o programador pode definir implementações para funções da classe, que funcionam como definições *default*, como no exemplo:

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

Com esta definição, o programador passa a ter que definir, em instâncias da classe *Eq*, apenas uma das funções (==) ou (/=). A implementação default será usada para a função que foi omitida. Por exemplo:

```
data Side = Left | Right

instance Eq Side where
         Right /= Left  = True
         Left  /= Right = True
         _     /= _     = False
```

Uma vez que a definição para o símbolo (==) foi omitida, a definição *default* é utilizada. Observe nesse exemplo que, como uma das funções utiliza a outra na sua definição, uma delas tem que ser obrigatoriamente definida.

Outro recurso relacionado com o uso de classes é a possibilidade de "derivar" automaticamente definições de funções. Essa característica é válida para algumas das classes da biblioteca padrão de *Haskell*, como: *Eq*, *Ord*, *Show*, *Enum* e *Read*. Quando definimos um novo tipo algébrico, podemos usar a cláusula *"deriving"* para obter declarações de instância de forma automática:

```
data Side = Left | Right deriving Eq
```

Nesse exemplo, definições para os símbolos (==) e (/=) são fornecidas sem que o programador tenha que implementá-las. Esse recurso não está disponível (diretamente na linguagem) para classes declaradas pelo programador.

2.2.1 Redução de Contexto

O conjunto de classes que definem as restrições aplicadas a uma determinada variável de tipo tem reflexos no processo de inferência de tipos. É claro que, quando em uma expressão ocorre um símbolo cuja suposição de tipo contém uma restrição, essa restrição deve ocorrer também no tipo inferido para a expressão. Dessa forma, uma lista de restrições — também chamada, na terminologia usada em *Haskell*, de *contexto* — é acumulada durante o processo de inferência de tipos. Em *Haskell*, após a inferência do tipo principal de uma determinada expressão, é realizado um processo, chamado de *redução de contexto*, que tem como objetivo simplificar (se possível) e verificar a validade (satisfazibilidade) dessa lista de restrições. A redução de contexto consiste em:

- Verificar se as restrições que ocorrem em tipos que foram instanciados para tipos concretos (ou seja, que não envolvem variáveis de tipo) são satisfeitas. Em caso positivo, a restrição é eliminada da lista de restrições; caso contrário, é reportado um erro de tipo. Por exemplo, considere que o tipo inferido para uma declaração envolva uma restrição *Eq Int*; como o tipo *Int* é uma instância da classe *Eq*, essa restrição é removida da lista de restrições do tipo sendo inferido, uma vez que a sobrecarga foi resolvida.
- Verificar se as restrições podem ser simplificadas, ou seja, se contextos que contenham restrições referentes a uma classe e sua superclasse aplicadas sobre uma mesma variável de tipo podem ser simplificados, eliminando a restrição que envolve a subclasse. Um exemplo dessa simplificação ocorre no caso da função *search* definida anteriormente (página 12).

– Verificar se não existe ambigüidade no tipo inferido. Uma expressão é considerada ambígua em *Haskell* se, em um tipo $\{C_1\ \alpha_1, ..., C_n\alpha_n\}.\tau$, onde existe uma variável de tipo α_i, $i \in \{1, ..., n\}$, que não ocorre em τ e também não ocorre no contexto de tipos no qual a expressão é tipada. Um exemplo clássico é o da expressão *show* (*read s*), onde *s* é uma cadeia de caracteres qualquer. As funções de biblioteca *show* e *read* têm tipos: $\forall\alpha.\{Show\ \alpha\}.\ \alpha \rightarrow String$ e $\forall\alpha.\{Read\ \alpha\}.\ String \rightarrow \alpha$, respectivamente. O tipo inferido para *show* (*read s*) seria $\forall\alpha.\{Show\ \alpha,\ Read\ \alpha\}.\ String$, que é um tipo ambíguo, pois não permite que as sobrecargas das funções *read* e *show* nessa expressão sejam resolvidas.

2.2.2 Classes de Tipos com Múltiplos Parâmetros

A generalização do conceito de classes de tipos, permitindo a definição dessas classes com múltiplos parâmetros, foi originalmente proposta por Wadler e Blott[58]. Considere o seguinte exemplo (transcrito de [58]), que usa classes com dois parâmetros para suporte a definições de funções sobrecarregadas *coerce*, para conversão de valores de um tipo em outro:

```
class Coerce a b where
      coerce::a → b

instance Coerce Int Float
      coerce = convertInttoFloat
```

Vários artigos incluiram, posteriormente, exemplos de aplicações envolvendo classes com múltiplos parâmetros [30, 14], mostrando que essa extensão aumenta significativamente a flexibilidade no uso do polimorfismo de sobrecarga. Embora várias implementações de *Haskell* incluam classes com múltiplos parâmetros, essa extensão ainda não foi incluída na definição oficial da linguagem, por requerer um tratamento adicional para evitar a ocorrência de ambigüidades (considerando a atual regra de ambiguidade usada em *Haskell*). Consideremos o exemplo da classe *Collects*, transcrito de [29]:

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

A variável de tipo a é usada para representar o tipo dos elementos da estrutura de dados ("coleção"), enquanto b é uma variável que representa a própria estrutura de dados. Podemos definir instâncias da classe *Collects* como, por exemplo:

- Listas e outras estruturas com um construtor aplicado a um tipo como, por exemplo, árvores. Nesse caso seria natural requerer que os tipos dos elementos da estrutura de dados fossem instâncias da classe *Eq*.
- Uma estrutura de dados que permita a localização de seus elementos por meio de uma função *hash*.

  Considere, por exemplo:

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

Nesse exemplo ocorre um problema com o tipo da função *empty*. De acordo com a regra de ambigüidade adotada em *Haskell*, essa função é considerada como ambígua, uma vez que o tipo a ela atribuído é:

$$empty :: \{Collects\ \alpha\ \beta\}.\ \beta$$

Isso ocorre porque em *Haskell* um tipo $\{C_1\ \alpha_1, ..., C_n\ \alpha_n\}.\ \tau$, no qual a variável α_i, $i \in \{1, ..., n\}$ ocorre na restrição e não ocorre no tipo τ, é considerado como ambíguo. Essa regra de ambigüidade foi definida para classes de tipos com um único parâmetro, e seu uso no caso de classes com múltiplos parâmetros tem apresentado problemas. Dependendo do contexto em que a função *empty* for utilizada, a sobrecarga poderia ser perfeitamente resolvida, como no caso da expressão:

```
[1] == empty
```

Um outro problema que ocorre com a classe *Collects*, mesmo removendo a declaração de *empty*, é que, pelo fato de não existir nenhuma declaração relacionando o tipo representado pela variável a com o tipo representado por b, a detecção de erros pode ser postergada. Uma alternativa é declarar a classe *Collects* como:

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

Com essa declaração o problema da ambigüidade na função *empty* é evitado e a relação entre os tipos é preservada. Por outro lado, a classe torna-se menos genérica, só podendo ser instanciada para tipos em que a coleção tenha a forma $c\ a$, com construtor c e elementos do tipo a. Isso exclui, por exemplo, a instância que utiliza a função *hash* do exemplo acima. Para manter a informação sobre a dependência dos tipos na declaração de classes, resolvendo desta forma alguns casos de ambigüidade (como no caso da função *empty*), assim como o problema de postergar a detecção de erros de tipo, foi introduzida na declaração das classes a possibilidade de especificar dependências entre as variáveis de tipos introduzidas nessas declarações. Esse recurso especifica as chamadas *dependências funcionais* [29] (entre as variáveis de tipo que são parâmetros da classe). A classe *Collects* pode ser declarada, usando dependências funcionais, da seguinte forma:

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

A dependência funcional $b \rightarrow a$ especifica que o tipo de a pode ser determinado a partir do tipo de b. Qualquer declaração que viole essa dependência é rejeitada. Dessa forma, as declarações a seguir não podem ocorrer juntas em um mesmo programa:

```
instance Collects Int [String] where ...
instance Collects Int [Int] where ...
```

Em extensões de *Haskell* com suporte a classes com múltiplos parâmetros, a especificação de dependências funcionais é utilizada na detecção de ambigüidade, sendo considerado ambíguo um tipo em que uma variável de tipo aparece somente na restrição e, além disso, não é "determinada" por uma dependência funcional. No entanto, a especificação de dependências funcionais fica a cargo do programador e deve ser determinada a priori, o que constitui um trabalho adicional, alheio aos propósitos originais de desenvolvimento de programas e, além disso, muitas vezes pode restringir as possibilidades de sobrecarga além do necessário.

Aperfeiçoamento dos Tipos Inferidos

Além de ajudar no processo de detecção de ambigüidades, a declaração de dependências funcionais também pode ser usada para simplificar o tipo inferido, em um processo chamado de *aperfeiçoamento de tipos*[2], que foi proposto por Mark Jones em [27]. Um bom exemplo é o da sobrecarga dos operadores aritméticos também proposto por Mark Jones em [29]. Esses operadores, a exemplo do operador de multiplicação (∗), são definidos em *Haskell* com tipo $\forall\alpha.\{Num\ \alpha\}.\ \alpha \to \alpha \to \alpha$, o que significa que o resultado e os dois argumentos têm sempre o mesmo tipo. Uma abordagem mais flexível poderia permitir argumentos com diferentes tipos, como nas declarações para o operador de multiplicação a seguir:

```
class Mul a b c | a b → c where
      (*):: a → b → c

instance Mul Int Int Int where ...
instance Mul Int Float Float where ...
instance Mul Float Int Float where ...
instance Mul Float Float Float where ...
```

Seria possível, com essa declaração de classe, sobrecarregar o operador de multiplicação para outros tipos de dados como, por exemplo, matrizes e vetores, implementando além da multiplicação de duas matrizes e dois vetores, a multiplicação dos mesmos por um escalar:

```
instance Mul a b c ⇒ Mul (Vec a) (Vec b) (Vec c) where ...
instance Mul a b c ⇒ Mul a (Vec b) (Vec c) where ...
instance Mul a b c ⇒ Mul (Mat a) (Mat b) (Mat c) where ...
instance Mul a b c ⇒ Mul a (Mat b) (Mat c) where ...
```

A dependência funcional $a\ b \to c$ informa que o tipo c pode ser determinado a partir dos tipos representados por a e b. Sem a informação sobre essa dependência ocorreriam problemas envolvendo ambigüidades em várias expressões simples como, por exemplo, a expressão: $(1 * 2) * 3$. Em um contexto onde as constantes 1, 2 e 3 têm tipo *Int*[3] essa expressão teria tipo:

```
(1 * 2) * 3 :: (Mul Int Int α, Mul α Int β) ⇒ β
```

[2] *improvement*

[3] As constantes 1, 2 e 3 são sobrecarregadas em *Haskell*, tendo tipo $\forall\alpha.\{Num\ \alpha\}.\ \alpha$.

Uma vez que a dependência funcional informa que o tipo de a deve ser determinado pelos tipos *Int* e b, essa expressão não é considerada ambígua. Ou seja, como nesse contexto existe uma instância da classe *Mul* com os dois primeiros parâmetros tendo o tipo *Int*, o tipo do terceiro parâmetro da primeira restrição que aparece na expressão (o tipo α) pode ser determinado como sendo *Int*, o que permite que o tipo de β também seja determinado como *Int*. Em *Haskell* esse processo de aperfeiçoamento dos tipos só ocorre se são especificadas dependências funcionais. Seja κ um conjunto de restrições aplicadas sobre um tipo principal, S uma substituição e $\lfloor\kappa\rfloor$ o conjunto de instâncias que satisfazem à restrição κ, definido por:

$$\lfloor\kappa\rfloor = \{S\kappa \mid \text{as sobrecargas em } S\kappa \text{ são resolvidas}\}$$

Então, uma substituição S_a é uma substituição que aperfeiçoa tipos em κ se $\lfloor\kappa\rfloor = \lfloor S_a\kappa\rfloor$ e se todas as variáveis de tipo envolvidas em S_a que não aparecem em κ são variáveis livres no contexto de tipos.

2.2.3 Instâncias Sobrepostas

Duas declarações de instância são ditas sobrepostas se os tipos que elas instanciam podem ser unificados. Por exemplo[30]:

```
class Foo a where
      describe :: a → String

instance Foo [a] where
      describe x = "List"
instance Foo [Char] where
      describe x = "String"
```

Ao se declarar $[Char]$ como uma instância da classe *Foo*, estamos sobrepondo esta declaração à declaração de $[a]$, que já abrangia listas de quaisquer tipos de elementos. Instâncias sobrepostas não podem ser declaradas, segundo a definição de *Haskell*, mas são aceitas nos principais compiladores disponíveis.

2.2.4 Recursão Polimórfica

A recursão polimórfica ocorre principalmente em funções definidas sobre *tipos não uniformes*, que são tipos nos quais a definição inclui um componente recursivo que não é idêntico ao tipo definido. Várias aplicações para esses tipos têm sido propostas [12, 45, 46, 5]. Um exemplo é o tipo de dados que representa uma árvore binária perfeitamente balanceada, definido por Chris Okasaki[46] como:

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

Este tipo não é uniforme, pois o componente recursivo $Seq(t, t)$ é diferente do tipo $Seq\ t$. Um tipo não uniforme freqüentemente apresenta algoritmos mais eficientes que suas contrapartes uniformes, como no exemplo a seguir:

```
length Nil = 0
length (Cons x s) = 1 + 2 * (length s)
```

Essa função determina o tamanho de uma seqüência em tempo $O(\log n)$, enquanto a função *length* usual sobre uma lista executa em tempo $O(n)$. Esta definição utiliza recursividade polimórfica, uma vez que pode receber um valor de tipo $Seq\ \alpha$ para qualquer tipo α, e retorna um inteiro, mas chama a si mesma como se tivesse tipo: $Seq\ (\alpha, \alpha) \rightarrow Integer$. Um algoritmo que não provê suporte a recursão polimórfica tentaria inferir o tipo desta função unificando os tipos α com (α, α), o que resultaria em um erro de tipo.

É possível converter um tipo de dados não uniforme em um tipo uniforme, quebrando o componente não uniforme do tipo em um novo tipo. Por exemplo, o tipo acima pode ser transformado em um tipo uniforme através das seguintes definições:

```
data Seq t    = Nil | Cons t (Elems t) (Seq t)
data Elems t = None | Pair (Elems t) (Elems t)
```

Entretanto, a versão não uniforme:

- é mais curta, utilizando apenas um tipo de dados, em vez de dois;
- o código é escrito usando menos construtores, o que pode torná-lo mais eficiente (pois não necessita de casamento de padrões para os construtores *Pair* e *Elems*);

– torna explícita a condição invariante de que existe um elemento simples após o primeiro construtor *Cons*, um par de elementos após o segundo, um par de par de elementos após o terceiro, e assim sucessivamente.

Linguagens de programação com suporte a polimorfismo paramétrico usam uma das duas seguintes abordagens para o tratamento de definições recursivas (*Haskell* e *SML* são exemplos de linguagens que usam a primeira abordagem, e *Mercury* [15] é um exemplo de linguagem que usa a segunda).

A primeira abordagem impõe uma restrição sobre definições recursivas, considerando que a função sendo definida não pode ser usada polimorficamente no contexto de sua própria definição. Em *Haskell*, a recursão polimórfica é permitida, mas somente quando o programador anota explicitamente o tipo polimórfico na definição da função. Em *Haskell*, a função *length* poderia ser declarada da seguinte forma (a anotação do tipo é obrigatória):

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

Em ambas as linguagens *Haskell* e *SML*, para que uma função possa ser utilizada com tipo polimórfico no mesmo contexto (*binding-group*) em que foi declarada, o *front-end* do compilador (ou interpretador) deve fazer um ordenamento topológico das definições, examinando o grafo de chamada dessas funções. Considere, por exemplo, as declarações abaixo:

```
map f xs    = [f x | x <- xs]
compList    = map not
squareList  = map (\x -> x * x)
```

Nesse exemplo, a função *map* deve ter seu tipo inferido antes, em um contexto anterior ao contexto onde serão inferidos os tipos de *compList* e *squareList*.

A segunda abordagem permite definições recursivas sem impor nenhum tipo de restrição, mas o algoritmo usa um limite de iteração configurado pelo usuário para parar o processo de inferência e rejeitar o programa quando esse limite é excedido.

Em [21, 34], foi provada a equivalência entre os problemas de tipabilidade no sistema de tipos de *Milner-Mycroft* (extensão do sistema de Damas-Milner com possibilidade de recursão polimórfica) e o problema da semi-unificação. Kfoury et al.[2] provaram a indecidibilidade do problema da semi-unificação, embora

ainda não se conheça nenhuma instância do problema para a qual a execução de algoritmos propostos[21, 55] não termine.

O problema da semi-unificação é uma generalização do problema da unificação [22]. Dados um conjunto de pares de tipos $\{(\tau_i, \tau_i')\}^{i=1..n}$, este problema consiste em decidir se existe uma substituição S e um conjunto de substituições: $\{S_1, S_2, ..., S_n\}$ tais que:

$$S_1 S\tau_1 = S\tau_1', \;\; S_2 S\tau_2 = S\tau_2', \ldots, S_n S\tau_n = S\tau_n'$$

Normalmente uma instância deste problema é representada na forma de um conjunto de inequações com índices:

$$\{\tau_1 \leq^1 \tau_1', \tau_2 \leq^2 \tau_2', ..., \tau_n \leq^n \tau_n'\}$$

2.3 A complexidade da Inferência de Tipos

Um algoritmo de inferência de tipos tradicional para o sistema de Damas-Milner transforma um termo M de λ-*cálculo* com declarações `let` em uma instância $\mathcal{L}_M$ do problema da unificação, de tal forma que a presença de variáveis polimórficas introduzidas por declarações `let` podem fazer com que o tamanho de $\mathcal{L}_M$ cresça de forma exponencial ao tamanho de M. Sem a ocorrência de variáveis polimórficas esse crescimento seria linear. Esse comportamento do algoritmo foi observado apenas após aproximadamente 20 anos de uso de *ML* e contraria as observações empíricas que demonstram a eficiência dos algoritmos de inferência. Isso pode ser explicado pelo fato desse comportamento exponencial ser verificado em expressões que apresentam várias declarações `let` aninhadas, o que normalmente não ocorre na prática. Um exemplo que demonstra este comportamento foi proposto por Michell Wand e independentemente por Peter Buneman [41](onde N é igual a $\lambda x.\, x$, ou um termo fechado semelhante):

$$\begin{array}{l} M \texttt{ = let } x_0 = N \texttt{ in} \\ \qquad \texttt{let } x_1 = (x_0, x_0) \texttt{ in} \\ \qquad\quad \ldots \\ \qquad\qquad \texttt{let } x_n = (x_{n-1}, x_{n-1}) \texttt{ in } x_n \end{array}$$

Nesse exemplo, como cada definição de x_i no momento do seu uso tem tipo polimórfico, suas variáveis de tipo serão substituídas por variáveis livres antes da

unificação. O tipo principal de M terá, assim, um número de variáveis de tipo que cresce exponencialmente em função de n. O mesmo não ocorre com uma definição similar que envolve apenas tipos monomórficos, como:

$$
\begin{aligned}
&M = \lambda x.(x, x)\\
&x_1 = M\ \ N\\
&x_2 = M\ \ x_1\\
&\dots\\
&x_n = M\ \ x_{n-1}
\end{aligned}
$$

Nesse caso, apesar do tamanho do tipo principal crescer exponencialmente em relação ao valor de n, o fato das variáveis que ocorrem em cada um dos termos da dupla não ocorrerem livres na expressão λ (o que determina seu tipo como monormórfico) garante que este tipo principal pode ser representado por um grafo acíclico com número de nós calculado linearmente em função de n. Dessa forma, algoritmos eficientes de unificação [47, 37] podem inferir o tipo principal destas expressões em tempo linear. Uma análise com provas relativas à complexidade do problema da inferência de tipos em *ML* pode ser encontrada, por exemplo, em [32, 35].

Em linguagens que estendem o sistema de Damas-Milner com suporte a sobrecarga impondo restrições aos tipos polimórficos, a inferência de tipos envolve a verificação da satisfazibilidade das restrições em um contexto de tipos. O problema da satisfazibilidade de restrições (*CS-SAT*) foi definido por Dennis Volpano e Goffrey Smith [57], como: *dado um contexto de tipos* Γ *e um conjunto de restrições* κ*, determinar se existe uma substituição* S *tal que* $\Gamma \vdash S\kappa$ *é provável, onde* $\Gamma \vdash \{o_i : \tau_i\}^{i=1..n}$ *é definido como:* $\Gamma \vdash o_i : \tau_i$*, é uma fórmula derivável no sistema de tipos da linguagem, para todo* $i \in \{1, \dots, n\}$.

Em [57] Dennis Volpano e Goffrey Smith apresentam uma redução do problema da correspondência de Post (*PCP*) para o problema da satisfazibilidade de restrições, provando que o problema *CS-SAT* é indecidível na presença de suposições de tipo onde ocorrem definições mutuamente recursivas.

2.4 *CS-SAT* e políticas de sobrecarga

Uma vez demonstrada a indecidibilidade do problema *CS-SAT*, várias políticas que restringem a sobrecarga de nomes vêm sendo propostas para evitar que a execução de compiladores ou interpretadores não termine. Alguns sistemas de tipos (como *System O* [44] e o sistema de tipos de *C++*, por exemplo) adotam políticas de sobrecarga independente do contexto, que eliminam o problema

de ambigüidade e tornam bastante simples a verificação da satisfazibilidade e resolução de sobrecargas. Em contrapartida, esses sistemas restringem muito as possibilidades de sobrecarga, principalmente no caso de linguagens com suporte a polimorfismo paramétrico e funções de ordem superior (i.e. funções que podem receber funções como parâmetro ou retornar funções).

Em outra direção, trabalhos iniciais relacionados a resolução de sobrecarga (*e.g.* [31, 58]) não levam em consideração o problema da satisfazibilidade de restrições. Em conseqüência disso, expressões nas quais o conjunto de restrições não são satisfeitas são consideradas como bem tipadas. Por exemplo, a expressão *True* `+` *True* é considerada como bem tipada no sistema definido em [58], em um contexto onde o símbolo `+` é sobrecarregado apenas para valores inteiros e de ponto flutuante.

Uma política de sobrecarga que elimina a recursividade nas restrições e impõe ainda outras fortes restrições, chamada *sobrecarga por construtores* (*overloading by constructor*), foi proposta por Greoffey Smith em [51, 52]. Essa política permite resolver *CS-SAT* em tempo polinomial, mas impossibilita definições sobrecarregadas que poderiam ocorrer normalmente em programas, como por exemplo:

$$\begin{aligned}\Gamma_* = \{&\texttt{+}:Int \to Int \to Int,\ \texttt{+}:Float \to Float \to Float,\\ &\texttt{*}:Int \to Int \to Int,\ \texttt{*}:Float \to Float \to Float,\\ &\texttt{*}:\forall\alpha.\{\texttt{*}:\alpha \to \alpha \to \alpha,\ \texttt{+}:\alpha \to \alpha \to \alpha\}.\ Matrix\ \alpha \to Matrix\ \alpha \to Matrix\ \alpha\}\end{aligned}$$

Esse exemplo foi apresentado por Volpano [56] como motivação para propor uma política de sobrecarga um pouco menos restritiva, chamada *sobrecarga paramétrica*. Essa forma de sobrecarga não permite restrições recursivas e torna *CS-SAT* decidível. O conjunto de suposições que obedece essa política de sobrecarga paramétrica pode ser definido indutivamente como a seguir:

1. O conjunto vazio é paramétrico.
2. Se A é paramétrico sem nenhuma suposição de tipo para x e $\sigma = \forall\alpha_1...\alpha_n.\kappa.\tau$ onde, para cada $o : \tau' \in \kappa$, o é sobrecarregado e τ' é a generalização mínima dos tipos de o, então $\{x : \sigma\} \cup A$ é paramétrico.
3. Se A é paramétrico sem nenhuma suposição para x e B é um conjunto da forma:

$$\begin{cases} x : \forall\alpha_{1_1}...\alpha_{1_n}.\kappa_1.\,\tau[C_1\ \alpha_{1_1}...\alpha_{1_n}/\alpha] \\ \vdots \\ x : \forall\alpha_{m_1}...\alpha_{m_n}.\kappa_m.\,\tau[C_m\ \alpha_{m_1}...\alpha_{m_n}/\alpha] \end{cases}$$

onde:

- $\forall\alpha.\tau$ é a generalização mínima dos tipos de x,
- $C_i \neq C_j$ para $i \neq j$,
- $o : \tau' \in \kappa_i$ implica que τ' é a generalização mínima dos tipos de o e, ou o é sobrecarregado em A ou $o = x$.

então $A \cup B$ é paramétrico.

Pela definição acima, em um conjunto de suposições válido segundo a política de sobrecarga paramétrica não é permitida a ocorrência de restrições mutuamente recursivas. Mais precisamente, se uma relação de dependência for definida como: *f depende de g se no conjunto de restrições do tipo de f aparece uma suposição de tipos para g*, então a política de sobrecarga paramétrica assegura que o fechamento transitivo dessa relação é antissimétrico. Conseqüentemente, recursividades mútuas como a do exemplo abaixo não são permitidas:

$$\begin{aligned}\Gamma \texttt{ = } \{ & f : Int \rightarrow Int\texttt{, } \; g : Int \rightarrow Int\texttt{,} \\ & f : \forall\alpha.\{g : \alpha \rightarrow \alpha\}.\, [\alpha] \rightarrow [\alpha]\texttt{,} \\ & g : \forall\alpha.\{f : \alpha \rightarrow \alpha\}.\, Tree\ \alpha \rightarrow Tree\ \alpha\ \}\end{aligned}$$

Apesar de ser um pouco mais flexível, essa política para sobrecarga ainda impõe consideráveis restrições, não permitindo, por exemplo, definições sobrecarregadas em que a generalização mínima tenha mais que uma variável de tipo. Volpano [56] mostra que para sobre esse sistema *CS-SAT* é *NP-difícil*.

O estilo de sobrecarga adotado pela linguagem *Haskell* é baseado na declaração de classes de tipos com uma única variável, como visto anteriormente. Foi demonstrado, por Tobias Nipkow e Christian Prehofer[43], que o problema *CS-SAT*, em um contexto que adote essa política, é um problema exponencial. Em [57] Volpano provou o mesmo resultado, usando o sistema de tipos definido por Geoffrey Smith[51].

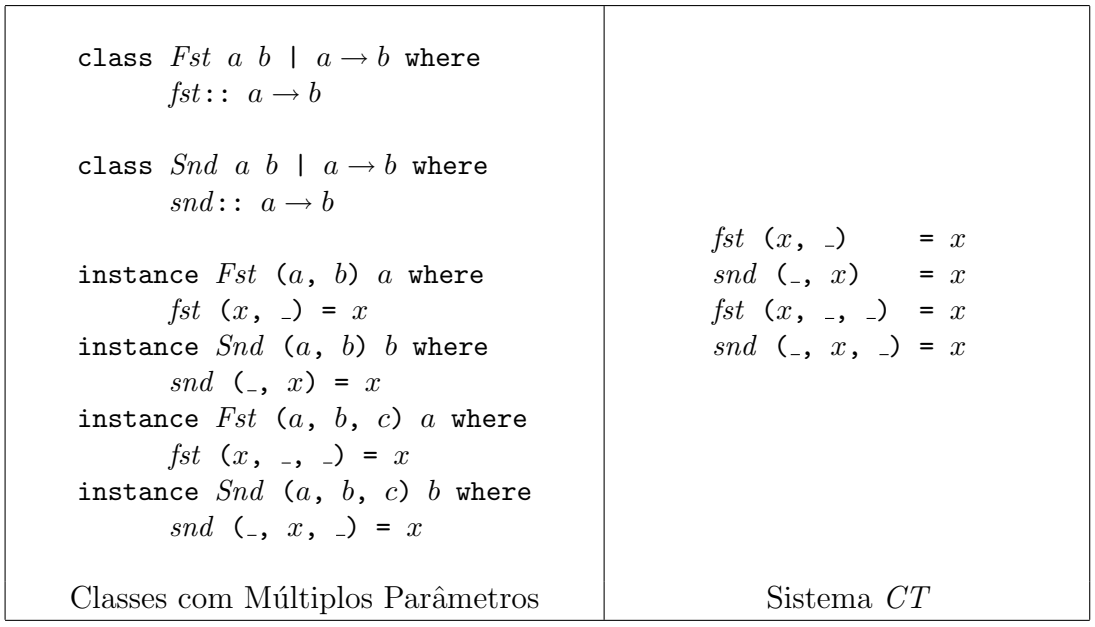

Figura 2.3: Comparação entre declarações sobrecarregadas em *Haskell* e em uma linguagem baseada no sistema *CT*

2.5 Motivação

A principal motivação desse trabalho é o estudo de um sistema de tipos que estenda o sistema de Damas-Milner com suporte a sobrecarga mas mantendo a simplicidade e flexibilidade desse sistema no suporte a inferência de tipos de expressões, com base na regra simples de que o tipo de uma expressão é determinado de acordo com o conjunto de declarações visíveis no contexto em que essa expressão ocorre.

O sistema de classes de tipos com múltiplos parâmetros, presente em algumas extensões de *Haskell*, embora permita expressar relações entre tipos de forma relativamente flexível e segura, muitas vezes obriga o programador a fazer várias declarações adicionais para que símbolos sobrecarregados possam ser definidos. Um exemplo simples dessa situação ocorre no caso da definição de funções sobrecarregadas para acesso ao primeiro e segundo elementos de duplas e triplas (*fst* e *snd*). A comparação entre as declarações necessárias em uma extensão de *Haskell* com suporte a classes com múltiplos parâmetros e em uma linguagem similar a *Haskell* que adote o sistema *CT* é apresentada na Figura 2.3. Nesse exemplo, são necessárias declarações de duas classes, uma relativa à sobrecarga da função *fst* e outra relativa à sobrecarga da função *snd*. Mesmo levando-se em conta que *Haskell* permite várias declarações contendo os tipos de símbolos sobrecarregados em uma única declaração de classe (um recurso que pode ser utilizado para diminuir o número de classes necessárias em programas), nem sempre as relações entre os tipos permitem o uso deste recurso. No caso

deste exemplo, as relações entre os tipos em questão impedem que as funções sejam declaradas como membros da mesma classe.

O comportamento exponencial, no pior caso, dos algoritmos de inferência de tipos em linguagens baseadas no sistema de Damas-Milner e, mais particularmente, em linguagens que adotam alguma extensão do sistema proposto por Damas-Milner para suporte a sobrecarga, somado ao fato de que, mesmo sendo aplicadas severas restrições sobre a política de sobrecarga de símbolos, o problema *CS-SAT* continua necessitando de algoritmos que têm comportamento exponencial no pior caso, constitui estímulo para desenvolvimento de um sistema de tipos mais flexível que imponha um mínimo de restrições à política de sobrecarga, com a adoção de um limite de iterações para tratar as situações onde a satisfazibilidade de restrições e a resolução de sobrecarga não possa ser decidida.

3 Sistema CT

Como mencionado anteriormente, o sistema de tipos CT [7] estende o sistema de Damas-Milner com suporte para sobrecarga de nomes. Os tipos das expressões são tipos polimórficos, onde cada tipo polimórfico pode conter um conjunto de restrições de tipo. Um conjunto de restrições de tipo κ é um conjunto de pares $o : \tau$, onde o é um nome (ou símbolo) sobrecarregado e τ é um tipo simples (i.e. um tipo não quantificado e no qual não ocorrem restrições de tipo).

Um tipo polimórfico com restrições é escrito na forma: $\forall\alpha_1...\alpha_n.\,\kappa.\,\tau$, onde $n \geq 0$. Um contexto de tipos Γ é um conjunto de suposições de tipo $x : \sigma$, onde x é uma variável (ou símbolo) e σ um tipo polimórfico com restrições. Cada definição sobrecarregada para um determinado símbolo introduz uma nova suposição de tipo para esse símbolo no contexto. O tipo principal de um símbolo sobrecarregado o é obtido a partir da generalização mínima (lcg) do conjunto de suposições de tipos para o em Γ.

A generalização mínima τ para um conjunto de tipos $\{\tau_1, ..., \tau_n\}$ é caracterizada pelas seguintes condições:

- existência de um conjunto de substituições S_i, para $i = 1, ..., n$ tal que $S_i\tau = \tau_i$;
- se existir um conjunto de substituições S'_i, para $i = 1, ..., n$ e um tipo e τ' tal que $S'_i\tau' = \tau_i$ então existe também uma substituição S tal que $S\tau' = \tau$.

A primeira condição expressa que τ é uma generalização do conjunto $\{\tau_1, ..., \tau_n\}$, e a segunda garante que τ é a generalização mínima. Um algoritmo para obtenção da generalização mínima de um conjunto de tipos é apresentado a seguir (Figura 3.2, página 31), nessa Seção. O uso da generalização mínima para obter o tipo principal de ume expressão não deve constituir uma surpresa, uma vez que o tipo principal é o tipo mínimo mas geral o suficiente para representar o conjunto de todos os tipos deriváveis para a expressão.

Vamos considerar, por exemplo, a função *insert* descrita no capítulo anterior e verificar como ela seria declarada em uma linguagem similar a *Haskell* que utilizasse o sistema de tipos CT. Lembramos mais uma vez que no sistema CT

não existe declaração de classes; quando desejamos sobrecarregar um símbolo, basta fazer uma nova declaração para o mesmo. Vamos supor que existam no contexto de tipos Γ as seguintes definições sobrecarregadas para os operadores de igualdade e "menor que":

$$\begin{aligned}
\texttt{(==)} &: Int \to Int \to Bool \\
\texttt{(==)} &: Char \to Char \to Bool \\
\texttt{(<)} &: Int \to Int \to Bool \\
\texttt{(<)} &: Char \to Char \to Bool
\end{aligned}$$

Definimos então a função *insert* que insere elementos de um tipo qualquer em uma lista:

$$\begin{aligned}
&insert\ x\ \texttt{[ ]} &&= \texttt{[}x\texttt{]} \\
&insert\ x\ \texttt{(}y\texttt{:}ys\texttt{)} &&= \texttt{if}\ x\ \texttt{==}\ y\ \texttt{then}\ y\texttt{:}ys\ \texttt{else}\ y\texttt{:(}insert\ x\ ys\texttt{)}
\end{aligned}$$

No momento em que o tipo da expressão x==y estiver sendo inferido, é verificado que (==) é um símbolo sobrecarregado, pois existem duas suposições de tipo para esse símbolo no contexto de tipos. Seu tipo principal é então obtido a partir da generalização mínima das duas definições presentes, que no caso é igual a $\alpha \to \alpha \to Bool$. Dessa forma, no contexto onde esse símbolo sobrecarregado está sendo usado, o tipo inferido para (==) é:

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

Essa restrição é adicionada ao tipo inferido para esta definição da função *insert*, uma vez que o símbolo sobrecarregado é usado em sua definição. Passamos a ter então um contexto Γ' com as seguintes suposições:

$$\begin{aligned}
\texttt{(==)} &: Int \to Int \to Bool \\
\texttt{(==)} &: Char \to Char \to Bool \\
\texttt{(<)} &: Int \to Int \to Bool \\
\texttt{(<)} &: Char \to Char \to Bool \\
insert &: \forall\alpha\texttt{.}\{\texttt{(==)}\texttt{:}\alpha \to \alpha \to Bool\}\texttt{.}\ \alpha \to [\alpha] \to [\alpha]
\end{aligned}$$

Continuando o exemplo, vamos sobrecarregar a função *insert* adicionando as seguintes definições:

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

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

Essa nova instância da função *insert* tem seu tipo inferido como:

$$\forall\alpha.\{\texttt{(==)}:\alpha \to \alpha \to Bool,\ \texttt{(<)}:\alpha \to \alpha \to Bool\}.\alpha \to Tree\ \alpha \to Tree\ \alpha$$

O tipo de *insert*, no contexto Γ_{insert} contendo as suposições em Γ' mais uma suposição de tipo correspondente a essa definição de *insert* para árvores, é inferido como:

$$\forall\alpha\forall\beta.\{insert\texttt{:}\ \alpha \to \beta\ \alpha \to \beta\ \alpha\}.\alpha \to \beta\ \alpha \to \beta\ \alpha$$

É importante notar que nenhuma informação sobre as restrições em relação aos símbolos (==) e (<) foram adicionadas à restrição da função *insert* nesse contexto. Quando uma das funções sobrecarregadas for selecionada, com base nos tipos requeridos no contexto onde a função é usada, a satisfazibilidade das restrições presentes no tipo suposto é verificada. Por exemplo, para a expressão *insert* `'a' [ ]`, primeiro é verificado se a restrição $\{insert : \alpha \to \beta\ \alpha \to \beta\ \alpha\}$ é satisfazível para listas de caracteres. Uma vez encontrada uma suposição de tipos que satisfaz esta restrição, é verificado se as restrições impostas a ela são também satisfeitas: no caso, a restrição $\{(\texttt{==}) : Char \to Char \to Bool\}$. Em qualquer caso, quando não se encontra nenhuma suposição de tipos no contexto que satisfaça às restrições, um erro de tipo é reportado.

Uma nova definição para um símbolo sobrecarregado não tem que ser uma instância da generalização mínima das definições anteriores. Uma nova definição pode ocasionar a mudança da generalização mínima para um tipo mais geral. Considerando os exemplos anteriores de definições de *insert*, podemos introduzir uma nova definição para a função *insert*, como a seguir:

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

$$\forall\alpha.\,\forall\beta.\,\forall\gamma.\,\{insert : \alpha \to \beta \to \gamma\}.\ \alpha \to \beta \to \gamma$$

α		β		γ
α	$\to$	$[\alpha]$	$\to$	$[\alpha]$
α	$\to$	$Tree\ \alpha$	$\to$	$Tree\ \alpha$
$\alpha \to \alpha \to Bool$	$\to$	α	$\to$	$[\alpha] \to [\alpha]$

$$\alpha = lcg(\alpha, \alpha, \alpha \to \alpha \to Bool)$$
$$\beta = lcg([\alpha], \alpha, \alpha)$$
$$\gamma = lcg([\alpha], Tree\ \alpha, [\alpha] \to [\alpha])$$

Figura 3.1: Inferência do tipo da função *insert* a partir da generalização minima de suas definições

Essa definição tem tipo principal $\forall\alpha.\,(\alpha \to \alpha \to Bool) \to \alpha \to [\alpha] \to [\alpha]$. Com essas três definições para a função *insert*, seu tipo seria inferido como mostrado na Figura 3.1.

A função *lcg* é definida na Figura 3.2. Fazemos uma simplificação ao considerar *lcg* como uma função (pois de fato *lcg* é uma relação, não uma função), escolhendo para isso como resultado de *lcg* qualquer representante da classe de equivalência entre tipos que são generalizações mínimas de um dado conjunto de tipos. Dois tipos são considerados equivalentes em um dado contexto de tipos se cada um pode ser obtido a partir do outro pela renomeação de variáveis de tipo que não ocorrem nesse contexto.

No sistema *CT*, o tipo de cada nome e de cada expressão é determinado de acordo com as suposições de tipo que fazem parte do contexto de tipos onde o nome ou expressão são usados, ou seja, de acordo com os nomes que são visíveis no contexto do programa onde o nome ou expressão ocorrem. Essa abordagem de *mundo fechado* e a possibilidade de uma linguagem baseada no sistema *CT* prover suporte também a uma abordagem de *mundo aberto* são discutidas com mais detalhes na Seção 3.5.

3.1 Definição Formal

Nesta Seção são definidas as regras de inferência do sistema de tipos e apresentado um algoritmo que infere os tipos para expressões de uma linguagem que consiste, basicamente, no núcleo da linguagem *ML* com a possibilidade adicional de introduzir definições sobrecarregadas no escopo de um programa. A sintaxe dessa linguagem e a definição da sintaxe das expressões de tipos do sistema *CT* são apresentadas na Figura 3. Meta-variáveis α e β são usadas como

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

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

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

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

Figura 3.2: Generalização Mínima

variáveis de tipo e representam um tipo concreto ou um construtor de tipos. C denota um construtor de tipos pertencente a um conjunto de construtores $\mathcal{C}$. Cada construtor de tipos $C\ \tau_1 ... \tau_n$ tem aridade n. O construtor de funções ($\rightarrow$) tem aridade 2, sendo normalmente escrito em notação infixada.

Para simplificar, variáveis ($x \in X$) são divididas em dois grupos distintos: variáveis ligadas por `let` ($o \in O$) e variáveis ligadas por λ-abstrações ($u \in U$). Constantes nessa linguagens são consideradas como sendo variáveis definidas em uma expressão `let` em um contexto global e que têm tipo fechado e sem nenhuma restrição de tipos. $\forall\overline{\alpha}.\kappa.\tau$ é usado para abreviar $\forall\alpha_1 ... \alpha_n.\kappa.\tau$, onde $n \geq 0$. $\forall\overline{\alpha}.\emptyset.\tau$ pode ser abreviado como $\forall\overline{\alpha}.\tau$. De forma similar, $\overline{\kappa.\tau}$ denota $\{\kappa_i.\tau_i\}^{i=1..n}$, onde $n \geq 0$. Um conjunto de suposições de tipo (possivelmente vazio) $\{x_i : \sigma_i\}^{i=1..n}$ é representado por meta-variáveis A ou Γ. Sendo $A = \{x_i : \sigma_i\}^{i=1..n}$, definimos: $dom(A) = \{x_i\}^{i=1..n}$, $A(x) = \{\sigma_i\}^{i=1..n}$ e $A \ominus x = A - \{x : \sigma_i\}^{i=1..n}$, e se $x \in U$ e $x : \sigma \in \Gamma$ então $\sigma = \tau$, para algum tipo simples τ. O conjunto de todas as variáveis de tipo livres em um tipo σ é usualmente denotado por $tv(\sigma)$. De forma similar são definidos $tv(\kappa)$ e $tv(A)$, considerando os tipos que ocorrem em κ e A, respectivamente. $tv(t_1 ... tn)$ é usado como abreviação para $tv(t_1) \cup ... \cup tv(t_n)$, e um tipo σ é dito *fechado* se $tv(\sigma) = \emptyset$.

$$
\begin{array}{lll}
\texttt{Expressões} & e ::= e\ e' \mid \lambda x.e \mid \texttt{let}\ o = e\ \texttt{in}\ e' & \\
\texttt{Programa} & p ::= e \mid \texttt{let}_o\ o = e\ \texttt{in}\ e' & \\
\\
\texttt{Tipos Simples} & \tau ::= C\ \tau_1 ... \tau_n \mid \alpha\ \tau_1 ... \tau_n \mid \alpha & (n \geq 0) \\
\texttt{Restrições} & \kappa ::= o : \tau \mid \kappa \cup \kappa' & \\
\texttt{Tipos} & \sigma ::= \tau \mid \kappa.\tau \mid \forall \alpha.\sigma &
\end{array}
$$

Figura 3.3: Sintaxe abstrata do sistema CT

Uma substituição S — uma função de variáveis de tipo em expressões de tipo — é também representada como uma função finita $S = \{(\alpha_i \mapsto \tau_i)\}^{i=1..n}$. $S \dagger \{(\alpha_i \mapsto \tau_i)\}^{i=1..n}$ denota uma substituição S' tal que $S'(\beta) = S(\beta)$ se $\beta \notin \{\alpha_i\}^{i=1..n}$ e $S'(\alpha_i) = \tau_i$, em caso contrário. id denota a função identidade, e $dom(S) = \{\alpha \mid S(\alpha) \neq \alpha\}$.

3.1.1 Satisfazibilidade

Um conjunto de restrições κ é satisfeito em um conjunto de suposições de tipo Γ se $\Gamma \vDash \kappa$ é provável segundo as regras apresentadas na Figura 4. Essa definição provê uma definição do problema *CS-SAT* independente do sistema de tipos. Essa definição depende apenas do conjunto de restrições e do conjunto de suposições de tipo dados como entrada para o problema[9]. Seja 2^Γ o conjunto potência de Γ, e *inst* o predicado correspondente à definição usual de *instância genérica* (veja e.g. [13, 24]), que pode ser formalizado como: $inst(\sigma, \kappa.\tau)$ é verdadeiro se $\sigma = \forall \bar{\alpha}.\kappa'.\tau'$ e $\kappa.\tau = (\kappa'.\tau')[\bar{\tau}/\bar{\alpha}]$ para algum $\bar{\tau}$.

Uma derivação de $\Gamma \vDash \kappa$ pode envolver a derivação de $\Gamma \vDash \kappa'$, onde κ' ocorre no tipo de uma suposição $o : \sigma$ que satisfaz a uma restrição $o : \tau \in \kappa$. Considere, por exemplo:

$$
\begin{aligned}
\Gamma_{Eq} = \{&(\texttt{==}) : Int \to Int \to Bool \\
&(\texttt{==}) : Char \to Char \to Bool \\
&(\texttt{==}) : \{\alpha \to \alpha \to Bool\}.[\alpha] \to [\alpha] \to Bool\}
\end{aligned}
$$

A derivação de $\Gamma_{Eq} \vDash \{[Int] \to [Int] \to Bool\}$ envolve provar $\Gamma_{Eq} \vDash \{Int \to Int \to Bool\}$.

Como outro exemplo de uso da regra (`sat`) considere o seguinte contexto de tipos:

$$\begin{aligned}\Gamma_f = \{&f : Int \to Int\\ &f : Int \to Float\\ &f : Float \to Float\}\end{aligned}$$

De acordo com essa regra, as restrições $\{f : Int \to Int\}$, $\{f : Int \to \beta\}$, $\{f : Int \to Float\}$, $\{f : \alpha \to \beta\}$ e $\{f : Float \to Float\}$ são satisfeitas em Γ_f, mas a restrição $\{f : Float \to \beta\}$ não é, uma vez que não é a generalização mínima de nenhum subconjunto das suposições de tipo para f em Γ_f. Isso reflete o fato de que se f for usado nesse contexto em uma expressão com um argumento do tipo *Float*, seu resultado tem que ser *Float*.

A política de sobrecarga adotada no sistema *CT* adota poucas restrições, permitindo que um conjunto maior de contextos sejam considerados como válidos, se compararmos com outras políticas citadas na Seção 2.4. A política usada no sistema *CT* não garante a decidibilidade do problema *CS-SAT*, mas a experiência com a implementação do algoritmo de inferência de tipos tem mostrado que ela funciona bem na prática; ou seja, a adoção de um limite de iteração, usado para rejeitar os casos que não podem ser decididos, tem se mostrado uma alternativa viável, sendo raros os casos em que esse limite é alcançado, na prática.

Um contexto Γ é considerado válido pela política de sobrecarga adotada no sistema de tipos *CT* se $\rho_{ct}(\Gamma)$ é verdadeiro, onde:

$$\rho_{ct}(\Gamma) = \Big(global(\Gamma) \text{ e } naoSobreposta(\Gamma) \text{ e } \Gamma \text{ é um contexto bem formado}\Big)$$

- $global(\Gamma) = ((o : \sigma) \in \Gamma$ e $\#A(o) > 1$ implica que $tv(\sigma) = \emptyset)$[1]
- $naoSobreposta(\Gamma) = (o$ é um símbolo sobrecarregado, $\{\sigma, \sigma'\} \subseteq \Gamma(o)$, $\sigma' \neq \sigma, \sigma = \forall\bar{\alpha}.\kappa.\tau, \sigma' = \forall\bar{\alpha}'.\kappa'.\tau', tv(\bar{\alpha}) \cap tv(\bar{\alpha}') = \emptyset)$ implica que $unificar(\{(\tau, \tau')\})$ falha
- um contexto Γ é bem formado se, para todo $(o : \forall\bar{\alpha}.\kappa.\tau) \in \Gamma$, $\Gamma \vDash \kappa$ é provável.

Na Figura 5, são mostrados alguns exemplos de contextos válidos e não válidos de acordo com a política de sobrecarga do sistema *CT*.

[1] A definição de *global* é um pouco mais geral do que o nome sugere, uma vez que uma definiçção em um escopo interno é permitida se não envolver variáveis ligadas a λ-expressões.

$$\frac{\begin{array}{c}\{\{o_i : \sigma_{ij}\}^{i=1..n}\}^{j=1..m} \subseteq 2^{\Gamma} \\ \text{para } i = 1, \ldots, n,\ j = 1, \ldots, m : inst(\kappa_{ij}.\tau_{ij}, \sigma_{ij}) \\ \text{para } j = 1, \ldots, m : \Gamma \models \bigcup_{i=1..n} \kappa_{ij} \\ lcg(\{\tau_i\}^{i=1..n}, \{\{\tau_{ij}\}^{i=1..n}\}^{j=1..m})\end{array}}{\Gamma \models \{o_i : \tau_i\}^{i=1..n}} \quad (\texttt{sat})$$

Figura 3.4: Satisfazibilidade de restrições

Contexto	$\rho_{ct}(A_i)$	Razão
$A_1 = \{\ o : Int,\quad o : Float,\quad o : \forall a.\,\{o : a\}.\,[a]\ \}$	V	$A_1 \models \{o : a\}$
$A_2 = \{\ o : Int,\quad o : Float,\quad o : \forall a.\,\{one : a\}.\,a\ \}$	F	não $naoSobreposta(A_2)$
$A_3 = \{\ o : Int,\quad o : Float,\quad o : \forall a.\,\{o : [a]\}.\,[a]\ \}$	F	$A_3 \not\models o : [a]$
$A_4 = \{\ o : Int,\quad o : Float,\quad o : \forall a.\,\{t : [[a]]\}.\,[a],$ $t : Int,\quad t : Float,\quad t : \forall a.\,\{o : a\}.\,[a]\ \}$	F	$A_4 \not\models t : [[a]]$
$A_5 = \{\ o : Int \to Int,$ $o : \forall a, b.\,\{o : a \to b\}.\,[a] \to b\ \}$	F	$A_5 \not\models o : a \to b$
$A_6 = \{o : Int,\quad o : \forall a.\,\{o : a\}.\,[a]\ \}$	V	$A_6 \models o : a$

Figura 3.5: A política de sobrecarga do sistema CT em exemplos

3.1.2 Simplificação

As restrições aplicadas a um tipo inferido podem ser simplificadas removendo as restrições relativas a símbolos para os quais a sobrecarga já foi resolvida, ou substituindo-as por uma mais "simples". Por exemplo, no contexto Γ_{Eq}, definido na Seção anterior, a restrição $\{(\texttt{==}) : Int \to Int \to Bool\}$ pode ser removida, e a restrição $\{[\alpha] \to [\alpha] \to Bool\}$ pode ser simplificada para $\{\alpha \to \alpha \to Bool\}$. Essa última simplificação leva em conta que a política de sobrecarga adotada no sistema CT não permite sobrecargas de símbolos com tipos sobrepostos. As simplificações de restrições são definidas pelas regras mostradas na Figura 3.6.

3.1.3 Sistema de Tipos

As definições de satisfazibilidade e de simplificação de restrições de tipos são usadas na formalização do sistema de tipos CT. As regras de inferência para esse sistema, considerando a linguagem cuja sintaxe livre de contexto foi definida na Figura 3.3, são mostradas na Figura 3.7. Para simplificar o entendimento, definições recursivas são omitidas. Para o tratamento desse tipo de definição, seria necessário adicionar uma regra que tratasse declarações feitas por meio do operador de ponto fixo, ou outra regra similar.

$$\frac{\kappa' = \{o_i : \tau_i \mid o_i : \tau_i \in \kappa \text{ e } tv(\tau_i) \neq \emptyset\}}{\Gamma \vDash \kappa \gg \kappa'} \quad \text{(simpl-1)}$$

$$\frac{\begin{array}{l}\text{para } i = 1..n \\ o_i : \tau_i \in \kappa \quad \kappa_i.\tau_i' \in \Gamma(o_i) \quad inst(\tau_i, \tau_i') \text{ não é válido}\end{array}}{\Gamma \vDash \kappa \gg \kappa} \quad \text{(simpl-2)}$$

$$\frac{\begin{array}{l}\text{para } i = 1..n, \\ o_i : \tau_i \in \kappa \quad \kappa_i.\tau_i' \in \Gamma(o_i) \quad inst(\tau_i, \tau_i') \quad \{o_{ij} : \tau_{ij}\}^{j=1..m} \in \kappa_i \\ \Gamma \vDash \bigcup_{i=1..n} \{o_{ij} : \tau_{ij}\}^{j=1..m} \gg \kappa'\end{array}}{\Gamma \vDash \kappa \gg \kappa'} \quad \text{(simpl-3)}$$

Figura 3.6: Regras para simplificação de restrições de tipo

Definimos que o predicado $gen(\sigma, \kappa.\tau)$ é verdadeiro se $\sigma = \forall\bar{\beta}.\,\kappa.\,\tau[\bar{\beta}/\bar{\alpha}]$, para algum $\bar{\beta}$, e $\bar{\alpha} = tv(\kappa.\tau)$. Também usamos $\overline{\overline{\kappa.\tau}}$ para representar o tipo σ tal que $gen(\sigma, \kappa.\tau)$. De forma similar, usamos $\bar{\bar{\kappa}}$, onde $\kappa = \{o_i : \tau_i\}^{i=1..n}$, para representar $\{o_i : \bar{\bar{\tau}}_i\}^{i=1..n}$, sendo $\bar{\bar{\kappa}}$ também escrito na forma $\{\|o_1 : \tau_1, ..., o_n : \tau_n\|\}$.

3.2 Inferência de Tipos

As funções *sat* e *simplificar*, definidas nas Figuras 3.10 e 3.11, são usadas no algoritmo de inferência de tipos para verificar a satisfazibilidade das restrições e realizar sua simplificação, respectivamente. $sat(\kappa, \Gamma)$ falha ou retorna uma substituição S tal que $\Gamma \vDash S\kappa$ pode ser provado e, para qualquer S' para o qual $\Gamma \vDash S'\kappa$ pode ser provado, existe uma substituição R tal que $S' = R \circ S$. A substituição retornada por *sat* é usada, no algoritmo de inferência de tipos, para aperfeiçoar o tipo de maneira similar à apresentada na Seção 2.2.2. A verificação da satisfazibilidade de um conjunto de restrições $\kappa = \{o_i : \tau_i\}^{i=1..n}$ em um contexto de tipos Γ envolve determinar o maior conjunto de suposições $\{o_i : \forall\bar{\alpha}.\kappa_i.\tau_i'\}^{i=1..n}$ em Γ tal que exista uma substituição que unifique cada τ_i com τ_i'. Chamamos esse conjunto de *conjunto-sat* de κ em Γ, sendo a função que retorna esse conjunto ($cSat$) definida na Figura 3.8. Para que κ seja satisfeito em Γ tem que existir, no *conjunto-sat* de cada $o_i : \tau_i \in \kappa$, pelo menos um $(o_i : \forall\bar{\alpha}. : \kappa' : \tau')$ tal que κ' seja também satisfeito em Γ.

$$\Gamma, \{x : \sigma_i\}^{i=1..n} \vdash x : \sigma \qquad \text{(VARo)}$$
$$\text{onde } \sigma = \begin{cases} \sigma_1 & \text{se } n = 1 \\ \{x : \tau\}.\tau & \text{se } n > 1, \text{onde } \tau = lcg(\{\sigma_i\}^{i=1..n}) \end{cases}$$

$$\frac{\Gamma \vdash e : \forall(\alpha_j)^{j=1..m}.\kappa.\tau \quad \Gamma \vDash \kappa' \quad \Gamma \vDash \kappa' \gg \kappa''}{\Gamma \vdash e : \kappa''.\tau'} \qquad \text{(INSTo)}$$
$$\text{onde } \kappa'.\tau' = (\kappa.\tau)[\tau_j/\kappa_j]^{j=1..m}$$

$$\frac{\Gamma \vdash \sigma}{\Gamma \vdash \forall\alpha.\sigma} \quad \alpha \notin tv(\Gamma) \qquad \text{(GENo)}$$

$$\frac{\Gamma, \{u : \tau'\} \vdash e : \kappa.\,\tau}{\Gamma \vdash \lambda u.\, e :\, \kappa.\,\tau' \to \tau} \qquad \text{(ABSo)}$$

$$\frac{\Gamma \vdash e_1 : \kappa_1.\,\tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \kappa_2.\,\tau_2 \quad \Gamma \models \kappa_1 \cup \kappa_2}{\Gamma \vdash e_1\, e_2 : \kappa_1 \cup \kappa_2.\,\tau_1} \qquad \text{(APPLo)}$$

$$\frac{\Gamma \vdash e_1 : \kappa_1.\,\tau_1 \quad \Gamma, \{o : \sigma_1\} \vdash e_2 : \kappa_2.\,\tau_2 \quad \Gamma \models \kappa_1 \cup \kappa_2}{\Gamma \vdash let\ o = e_1\ in\ e_2 : \kappa_1 \cup \kappa_2.\,\tau_2} \; gen(\kappa_1.\,\tau_1, \sigma_1, \Gamma) \qquad \text{(LETo)}$$

$$\frac{\Gamma \vdash e_1 : \kappa_1.\,\tau_1 \quad \Gamma; \{o : \sigma_1\} \vdash p : \kappa_2.\,\tau_2 \quad \Gamma \models \kappa_1 \cup \kappa_2}{\Gamma \vdash let_o\ o = e_1\ in\ p : \kappa_1 \cup \kappa_2.\,\tau_2} \; gen(\kappa_1.\,\tau_1, \sigma_1, \Gamma) \qquad \text{(LETOo)}$$

Figura 3.7: Sistema *CT*

Uma Solução para *CS-SAT*

$sat(\kappa, \Gamma)$ é definido como $sats(\kappa, \Gamma, \varkappa, \emptyset)$, onde $\varkappa$ é um inteiro positivo que define um limite de iterações usado para terminar a execução do algoritmo, nos casos em que a satisfazibilidade não pode ser resolvida. O último parâmetro na chamada da função *sats*, que inicialmente é definido como um conjunto vazio, contém as restrições que já foram verificadas (considerando um determinado ramo da árvore de chamadas recursivas a *sats*).

Uma chamada a $sats(\{o_i : \tau_i\}^{i=1..n}, \Gamma, \varkappa, \kappa)$ com $cSat(\{o_i : \tau_i\}^{i=1..n}, \Gamma) = \{(\{o_i : \kappa_{ij}.\tau ij^{i=1..n}, S_j\}^{j=1..m})\}$ retorna a m (possivelmente zero) chamadas recursivas a *sats*. Em cada uma dessas chamadas recursivas é testado se as restrições que estão sendo verificadas estão caminhando na direção das folhas da árvore (*downward occurrence*), caso contrário é testado se estão formando um laço (*loop occurrence*). No segundo caso a verificação da satisfazibilidade de uma restrição envolve uma nova verificação dela mesma em um sub-ramo da árvore de chamadas. Esse processo é realizado removendo, do contexto passado

$$
\begin{array}{ll}
cSat(\emptyset) & = \{(\emptyset, id)\} \\
cSat(\{o:\tau\},\Gamma) & = \{(\{o:\kappa.\,\tau'\}, S) \mid\ o:\forall\bar{\alpha}.\,\kappa.\,\tau' \in \Gamma,\ \bar{\alpha}\cap tv(\tau)=\emptyset, \\
 & \qquad\qquad unificar((\tau,\tau')) = S\}
\end{array}
$$

$$
\begin{array}{l}
cSat(\{o:\tau\}\cup\kappa,\Gamma)\ = \\
\quad \texttt{let}\ \{(\{o:\kappa_i.\,\tau_i\}, S_i)\}^{i=1..n} = cSat(\{o:\tau\},\Gamma) \\
\qquad\quad \{(\Gamma_{ij}, S_{ij})\}^{j=1..m_i} \qquad = cSat(\{o:S_i\tau \mid o:\tau\in\kappa\},\Gamma),\ \texttt{for}\ i=1..n \\
\quad \texttt{in}\ \{(\Gamma_{ij}\cup\{o:\kappa_i.\,\tau_i\}, S_{ij}\circ S_i)\}^{i=1..n, j=1..m_i}
\end{array}
$$

Figura 3.8: *Conjunto-Sat*

$$\kappa' \lessdot \bar{\bar{\kappa}} = \big(\exists (o:\tau)\in\kappa' \mid \tau \lessdot \kappa(o)\big)$$

$\tau \lessdot \emptyset$ é válido, $\quad \tau \lessdot \{\tau'\} = \big(\tau \nsim \tau'$ ou $\tau < \tau'\big)$,
$\tau \lessdot \mathbb{T} = \big(\tau \lessdot \{\tau'\},$ para algum $\tau' \in \mathbb{T}\big)$

$(C\,\tau_1 \ldots \tau_n)\ \nsim (C'\,\tau'_1 \ldots \tau'_m) =$
$\big(C \neq C'$ ou $\tau_i \nsim \tau'_i$, para algum $i \in \{1,\ldots,n\}\ \big)$
$\tau \nsim \tau'$ não é válido, caso contrario

$$\bar{\bar{\kappa}} \leq \kappa' = \big(\exists (o:\tau)\,\bar{\in}\,\bar{\bar{\kappa}} \mid \tau \leq \kappa(o)\big) \quad (o:\tau)\,\bar{\in}\,\bar{\bar{\kappa}} = (o:\bar{\bar{\tau}} \in \bar{\bar{\kappa}})$$

$$\tau \leq \mathbb{T} = \big(\exists\ \tau' \in \mathbb{T} \mid \tau \leq \tau'\big)$$

Figura 3.9: Relações entre Restrições e Tipos

às próximas chamadas, a suposição de tipo que originou o laço. As relações que definem restrições caminhando em direção as folhas e restrições formando laço são definidas na Figura 3.9.

Veja o exemplo abaixo, que considera o contexto $A_6 = \{o : Int,\ o : \forall.\{o:\alpha\}.[\alpha]\}$ apresentado na Figura 5 e a seguinte chamada à função *sats*:

$$sats(\{o:\alpha\}, A_6, \varkappa, \emptyset)$$

Temos:

$$
\begin{array}{rcl}
cSat(\{o:\alpha\},\ A_6) & = & \{(\{o:Int\},\ \{\alpha\mapsto Int\}), \\
 & & \quad (\{o:\{o:\alpha'\}.[\alpha']\},\ \{\alpha\mapsto[\alpha']\})\}
\end{array}
$$

$$sat\,(\kappa,\Gamma) = \bigcap sats\big(\{\kappa,\Gamma,\varkappa,\emptyset\big)$$

$$sats(\emptyset,\Gamma,\varkappa,\bar{\bar{\kappa}}) = id$$
$$sats\left(\{o_i:\tau_i\}^{i=1..n},\Gamma,\varkappa,\bar{\bar{\kappa}}\right) =$$
$$\texttt{let } \left\{\left(\{o_i:\kappa_{ij}.\,\tau_{ij}\}^{i=1..n},S_j\right)\right\}^{j=1..m} = cSat\big(\{(o_i:\tau_i)\}^{i=1..n},\Gamma\big)$$
$$\texttt{in if } m=0 \texttt{ then } \textit{falha} \texttt{ else}$$
$$\texttt{let para } j=1,\ldots,m\texttt{:}$$
$$\kappa_j = \textstyle\bigcup_{i=1,\ldots,n} S_j\kappa_{ij}\texttt{,}$$
$$(\Gamma_j,\varkappa_j) =$$
$$\textit{(teste de ocorrência em direção às folhas)}$$
$$\texttt{if } \kappa_j \lessdot \bar{\bar{\kappa}} \texttt{ then } (\Gamma,\varkappa) \texttt{ else}$$
$$\textit{(teste de ocorrência formando laço)}$$
$$\texttt{if } \bar{\bar{\kappa}} \leq \kappa_j \texttt{ then } (\Gamma \ominus \{o_i:\kappa_{ij}.\,\tau_{ij}\}^{i=1..n},\varkappa) \texttt{ else}$$
$$\texttt{if } \varkappa > 0 \texttt{ then } (\Gamma,\varkappa-1) \texttt{ else } \textit{falha}$$
$$\bar{\bar{\kappa_0}} = \bar{\bar{\kappa}} \oplus \{o_i:\tau_i\}^{i=1..n}$$
$$\mathbb{S} = \left\{S'_j \circ S_j \mid S'_j = sats(\kappa_j,\Gamma_j,\varkappa_j,\bar{\bar{\kappa_0}})\right\}^{j=1..m}$$
$$\texttt{in if } \mathbb{S}=\emptyset \texttt{ then } \textit{falha} \texttt{ else } \mathbb{S}$$

Figura 3.10: *sat*

$$\begin{array}{lll}
simplificar(\kappa,\Gamma) & \texttt{=} & simpl(\kappa,\Gamma,\emptyset)\\[2ex]
simpl(\emptyset,\Gamma,\kappa_0) & \texttt{=} & \emptyset\\
simpl(\{o:\tau\},\Gamma,\kappa_0) & \texttt{=} & \texttt{if } tv(\tau)=\emptyset \texttt{ then } \emptyset\\
 & & \quad\texttt{else if } o:\tau\in\kappa_0 \texttt{ then } \{o:\tau\}\\
 & & \quad\texttt{else if existe } \forall\bar{\alpha}.\kappa'.\tau'\in\Gamma(o)\\
 & & \qquad\qquad\text{tal que } S\tau'=\tau\texttt{,} \text{ para algum } S\\
 & & \qquad\texttt{then } simpl(S\kappa',\Gamma,\kappa_0\cup\{o:\tau\})\\
 & & \qquad\texttt{else } \{o:\tau\}\\
simpl(\{o:\tau\}\cup\kappa,\Gamma,\kappa_0) & \texttt{=} & simpl(\{o:\tau\},\Gamma,\kappa_0)\cup simpl(\kappa,\Gamma,\kappa_0)
\end{array}$$

Figura 3.11: *simplificar*

$$\Gamma_0 \vdash^A x : \kappa.\tau \qquad \text{(VARo)}$$

$$\text{onde } \kappa.\tau = \begin{cases} \alpha & \text{se } \Gamma_0(x) = \emptyset, \text{ onde } \alpha \text{ é uma var. de tipo livre} \\ \kappa'\tau' & \text{se } \Gamma_0(x) = \forall\bar{\alpha}.\kappa'.\tau', \text{ onde} \\ & \kappa' = \mathit{simplificar}(\kappa[\bar{\beta}/\bar{\alpha}], \Gamma_0) \\ & \tau' = \tau[\bar{\beta}/\bar{\alpha}], \text{ onde } \bar{\beta} \text{ são vars. de tipos livres} \\ \{x : \tau''\}.\tau'' & \text{se } \Gamma_0(x) = \{\sigma_i\}^{i=1..n} \text{ e } n > 1 \\ & \text{onde } \tau'' = \mathit{lcg}(\{\sigma_i\}^{i=1..n}) \end{cases}$$

$$\frac{\Gamma_0, \{u : \alpha\} \vdash^A e : (\kappa.\,\tau, \Gamma)}{\Gamma_0 \vdash^A \lambda u.\, e :\ (\kappa.\,\tau' \to \tau, \Gamma \ominus u)} \qquad \text{(ABSo)}$$

onde $\tau' = \Gamma(u)$ e α é uma variável livre

$$\frac{\Gamma_0 \vdash^A e_1 : (\kappa_1.\,\tau_2 \to \tau_1, \Gamma_1) \quad \Gamma_0 \vdash^A e_2 : (\kappa_2.\,\tau_2, \Gamma_2)}{\Gamma_0 \vdash^A e_1\, e_2 : (\kappa.\tau, \Gamma)} \qquad \text{(APPLo)}$$

$$\begin{array}{lll} \text{onde} & S = \mathit{unificar}(\varepsilon(\Gamma_1, \Gamma_2) \cup (\tau_1, \tau_2)) & \alpha \text{ é uma variável livre} \\ & S_\triangle = \mathit{sat}(S\kappa_1 \cup S\kappa_2, \Gamma_1 \cup \Gamma_2) & \Gamma = S_\triangle(S\Gamma_1 \cup S\Gamma_2) \\ & \kappa = \mathit{simplificar}(S_\triangle(S\kappa_1 \cup S\kappa_2), \Gamma_1 \cup \Gamma_2) & \tau = S_\triangle S\alpha \end{array}$$

$$\frac{\Gamma_0 \vdash^A e_1 : (\kappa_1.\,\tau_1, \Gamma_1) \quad S_0\Gamma_0, \{o : \sigma_1\} \vdash^A (e_2 : \kappa_2.\,\tau_2, \Gamma_2)}{\Gamma_0 \vdash^A \mathit{let}\ o = e_1\ \mathit{in}\ e_2 : (\kappa.\,\tau, \Gamma)} \qquad \text{(LETo)}$$

$$\begin{array}{lll} \text{onde} & S_0 = \mathit{unificar}(\varepsilon(\Gamma_0, \Gamma_1)) & \sigma_1 = \mathit{gen}(\kappa_1.\,\tau_1, \Gamma) \\ & S = \mathit{unificar}(\varepsilon(\Gamma_1, \Gamma_2)) & \tau = S_\triangle S\tau_2 \\ & S_\triangle = \mathit{sat}(S\kappa_1 \cup S\kappa_2, \Gamma_1 \cup \Gamma_2) & \\ & \kappa = \mathit{simplificar}(S_\triangle(S\kappa_1 \cup S\kappa_2), \Gamma_1 \cup \Gamma_2) & \\ & \Gamma = S_\triangle(S\Gamma_1 \cup (S\Gamma_2 - \{o : S\sigma_1\})) & \end{array}$$

$$\frac{\Gamma_0 \vdash^A e_1 : (\kappa_1.\,\tau_1, \Gamma_1) \quad S_0\Gamma_0; \{o : \sigma_1\} \vdash^A (p : \kappa_2.\,\tau_2, \Gamma_2)}{\Gamma \vdash^A \mathit{let}_o\ o = e_1\ \mathit{in}\ p : \kappa_1 \cup \kappa_2.\,\tau_2} \qquad \text{(LETOo)}$$

$$\begin{array}{lll} \text{onde} & S_0 = \mathit{unificar}(\varepsilon(\Gamma_0, \Gamma_1)) & \sigma_1 = \mathit{gen}(\kappa_1.\,\tau_1, \Gamma) \\ & S = \mathit{unificar}(\varepsilon(\Gamma_1, \Gamma_2)) & \tau = S_\triangle S\tau_2 \\ & S_\triangle = \mathit{sat}(S\kappa_1 \cup S\kappa_2, \Gamma_1 \cup \Gamma_2) & \\ & \kappa = \mathit{simplificar}(S_\triangle(S\kappa_1 \cup S\kappa_2), \Gamma_1 \cup \Gamma_2) & \\ & \Gamma = S_\triangle(S\Gamma_1 \cup (S\Gamma_2 - \{o : S\sigma_1\})) & \end{array}$$

Figura 3.12: Algoritmo de inferência de tipos

onde α' é uma nova variável de tipo (que não ocorre livre em A_6). Como $o : \{o : \alpha'\}.[\alpha']$ satisfaz à restrição $o : \alpha$, a satisfazibilidade de $\{o : \alpha'\}$ tem que ser agora verificada, por meio de uma nova chamada à função *sats*. Como $\{o : \alpha'\} \lessdot \emptyset$, a chamada à função *sats* é dada por:

$$sats(\{o : \alpha'\}, A_6, \varkappa, \|o : \alpha\|)$$

Nessa chamada, é verificado se ocorre ou não um laço, por meio do teste $\{\|o : \alpha\|\} \leq \{o : \alpha'\}$. A próxima chamada recursiva a *sats* é, então:

$$sats(\{o : \alpha''\}, A'_6, \varkappa, \{\|o : \alpha\|\})$$

onde $A'_6 = A_6 - \{\forall\alpha.\{o : \alpha\}.[\alpha]\}$

Para essa nova chamada o *conjunto-sat* é dado por: $cSat(o : \alpha'', A'_6) = \{(o : Int, \{\alpha'' \mapsto Int\})\}$.

O resultado de $sats(\{o : \alpha''\}, A'_6, \varkappa, \{\|o : \alpha\|\})$ é então dado pela substituição $\{\alpha'' \mapsto Int\}$. Essa substituição é retornada para a chamada anterior, $sats(\{o : \alpha\}, A_6, \varkappa, \|o : \alpha\|)$, originando o seguinte conjunto de substituições:

$$\mathbb{S} = \{\{\alpha' \to Int\}, \{\alpha' \mapsto [Int]\}\}$$

A substituição $\bigcap \mathbb{S} = \{\alpha \to \beta\}$, onde β é uma variável de tipo livre, é retornada então como resultado de $sats(\{o : \alpha\}, A_6, \varkappa, \emptyset)$.

Em situações nas quais as restrições não caminham em direção às folhas da árvore e não ocorre um laço entre as restrições de tipo cuja satisfazibilidade está sendo verificada, é usado um limite predefinido de iterações para interromper o processo de verificação da satisfazibilidade, garantindo assim a sua terminação. Em testes realizados com a implementação do algoritmo, usando como entrada código similar a implementações reais feitas em *Haskell*, esse limite não foi necessário em nenhuma situação. O exemplo abaixo, onde o limite é necessário, nos foi apresentado por Martin Sulzmann:

$$\begin{aligned}\Gamma_T = \{\ & o : Int \to Bool,\\ & o : Char \to Int,\\ & o : \forall a, b.\, \{o : a \to b\}.\, T^2\ a \to b\}\end{aligned}$$

onde T^i é uma abreviação para o tipo composto por sucessivas aplicações do construtor de tipos T, sendo i seu número de ocorrências. Para um conjunto de restrições: $\kappa = \{o : \alpha \to T\alpha\}$, $sat(\kappa, \Gamma_T)$ não termina se um limite $\varkappa$ não for usado para interromper o processo:

$$
\begin{aligned}
& sat(\kappa, \Gamma_t) \\
& = sats(\kappa, \Gamma_t, \varkappa, \emptyset)\}) \\
& = sats(\{o : a_1 \to T^3 a_1\}, \Gamma, \varkappa - 1, \bar{\bar{\kappa}}) \circ S_1 \\
& \quad \text{onde } S_1 = \{\alpha \mapsto T^2 a_1, b_1 \mapsto T^3 a_1\} \\
& = sats(\{o : a_2 \to T^5 a_2\}, \Gamma, \varkappa - 2, \bar{\bar{\kappa}} \cup \|o : a_1 \to T^2 a_1\|) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \circ S_2 \circ S_1 \\
& \quad \text{onde } S_2 = \{a_1 \mapsto T^2 a_2, b_2 \mapsto T^5 a_2\} \\
& = \ldots \text{ (executa indefinidamente se } \varkappa \text{ não for testado)}
\end{aligned}
$$

Em algumas situações, a introdução do limite $\varkappa$ para o número máximo de chamadas recursivas a *sats* pode fazer com que uma expressão bem tipada seja recusada. Para ilustrar essa situação, consideremos uma instância do problema da satisfazibilidade de restrições obtido através da redução de uma instância do problema da correspondência de Post (*PCP*): existe uma seqüencia de inteiros $i_1, i_2..i_m$, $m \geq 1$, tal que $x_{i_1} x_{i_2}..x_{i_m} = y_{i_1} y_{i_2}..y_{i_m}$ onde $x_1 = C_1 C_0$, $x_2 = C_0 C_1 C_1$, $x_3 = C_1 C_0 C_1$, $y_1 = C_1 C_0 C_1$, $y_2 = C_1 C_1$ e $y3 = C_0 C_1 C_1$?

Esse exemplo e sua redução a *CS-SAT* foram apresentados por Dennis Volpano e Geoffrey Smith em [57]. Um exemplo similar é apresentado por Geoffrey Smith em [51], mas nesse caso a redução a *CS-SAT* envolve suposições de tipo contendo tipos sobrepostos, o que não é aceito pela política de sobrecarga do sistema *CT*. Dados os contexto de tipos Γ_p e o conjunto de restrições κ_p abaixo, as restrições κ_p são satisfeitas em Γ_p, então, se existe uma solução para essa instância de *CS-SAT* existe solução para a instância de *PCP* correspondente:

$$
\begin{aligned}
\Gamma_p = \{ & \\
& p : \ (C_1 \to C_0 \to C) \to (C_1 \to C_0 \to C_1 \to C) \to C'_1 \\
& p : \ (C_0 \to C_1 \to C_1 \to C) \to (C_1 \to C_1 \to C) \to C'_2 \\
& p : \ (C_1 \to C_0 \to C_1 \to C) \to (C_0 \to C_1 \to C) \to C'_3 \\
& p : \ \forall \alpha, \beta, \gamma. \{p : \alpha \to \beta \to \gamma\}. \\
& \qquad (C_1 \to C_0 \to \alpha) \to (C_1 \to C_0 \to C_1 \to \beta) \to (C'_1 \to \gamma) \\
& p : \ \forall \alpha, \beta, \gamma. \{p : \alpha \to \beta \to \gamma\}. \\
& \qquad (C_0 \to C_1 \to C_1 \to \alpha) \to (C_1 \to C_1 \to \beta) \to (C'_2 \to \gamma) \\
& p : \ \forall \alpha, \beta, \gamma. \{p : \alpha \to \beta \to \gamma\}. \\
& \qquad (C_1 \to C_0 \to C_1 \to \alpha) \to (C_0 \to C_1 \to \beta) \to (C'_3 \to \gamma) \ \}
\end{aligned}
$$

onde C, C_0, C_1, C'_1 e C'_2 são construtores de tipos de aridade 0.

Sendo $\kappa_p = \{p : \alpha \to \alpha \to \beta\}$, temos que $sat(\kappa_p, \Gamma_p)$ falha se e somente se $\varkappa < 1$, uma vez que:

$$satset(\{(p : a \to a \to b, \Gamma_p)\}) = \{(\{p : \kappa.\,\tau\}, S)\}$$

onde

$$\begin{aligned}\kappa.\,\tau =& \{p : \alpha_1 \to \beta_1 \to \gamma_1\}.\,(C_1 \to C_0 \to a_1) \to \\ & (C_1 \to C_0 \to C_1 \to \beta_1) \to (C'_1 \to c_1)\}\end{aligned}$$

$$\begin{aligned}S = \big\{\big(\ & a_1 \mapsto C_1 \to \beta_1, b \mapsto (C'_1 \to \gamma_1), \\ & a \mapsto (C_1 \to C_0 \to C_1 \to b_1)\big)\big\}\end{aligned}$$

A chamada recursiva a *sats* é dada por (o valor de $\varkappa$ é decrementado uma vez que nem a relação $\{(C_1 \to \beta_1) \to \beta_1 \to \gamma_1\} \lessdot \{\alpha \to \alpha \to \beta\}$ e nem a relação $\{\alpha \to \alpha \to \beta\} < \{(C_1 \to \beta_1) \to \beta_1 \to \gamma_1\}$ são satisfeitas):

$$sats(\{p : (C_1 \to \beta_1) \to \beta_1 \to \gamma_1\}, \Gamma_p, \varkappa - 1, \|p : \alpha \to \alpha \to \beta\|)$$

Essa chamada envolve a verificação do conjunto de suposições que satisfaz as restrições através da chamada a *cSat*:

$$cSat(\{p : (C_1 \to \beta_1) \to \beta_1 \to \gamma_1\}, \Gamma_p)$$

que retorna:

$$\begin{aligned}\big\{\ & (p : (C_1 \to C_0 \to C_1 \to C) \to (C_0 \to C_1 \to C) \to C'_3, S_1), \\ & (p : \{p : \alpha_2 \to \beta_2 \to \gamma_2\}.\ (C_1 \to C_0 \to C_1 \to \alpha_2) \to \\ & \qquad\qquad (C_0 \to C_1 \to \beta_2) \to (C'_3 \to \gamma_2), S_2)\ \big\}\end{aligned}$$

onde: $S_1 = id \dagger \{(\beta_1 \mapsto C_0 \to C_1 \to C, \gamma_1 \mapsto C'_3)\}$ e $S_2 = id \dagger \{(\alpha_2 \mapsto \beta_2, \beta_1 \mapsto C_0 \to C_1 \to \beta_2, \gamma_1 \mapsto (C'_3 \to c_2))\}$.

O primeiro resultado implica em uma nova chamada a *sats* com um conjunto de restrições vazio, cujo resultado é a substituição identidade. O segundo envolve outra chamada a *sats*:

$$sats(\{p : \beta_2 \to \beta_2 \to \gamma_2,\ \Gamma_p - \{p : \forall\alpha, \beta, \gamma.\, \{p : \alpha \to \beta \to \gamma\}.\\ (C_1 \to C_0 \to C_1 \to \alpha) \to (C_0 \to C_1 \to \beta) \to (C_3' \to \gamma)\}\},\\ \varkappa - 1, \|p : \alpha \to \alpha \to \beta,\ p : (C_1 \to \beta_1) \to \beta_1 \to \gamma_1\|)$$

O resultado final é dado por:

$$\bigcap\{S \circ S_1 \circ id, S \circ S_2 \circ S_3 \circ id\}$$

onde $S_3 = \{\alpha_3 \mapsto C_1 \to \beta_3,\ \beta_2 \mapsto C_1 \to C_0 \to C_1 \to \beta_3,\ \gamma_2 \mapsto C_1' \to \gamma_3\}$.

Para qualquer valor de ℓ, pode-se encontrar um conjunto de restrições κ_ℓ tal que $sat(\kappa_\ell, \Gamma_p)$ falha para $\varkappa = \ell$. Por exemplo se $\ell = \varkappa = 1$, temos $\kappa_\ell = \{p : (C_0 \to C_1 \to \alpha) \to (C_1 \to C_1 \to \alpha) \to \beta)$; se $\ell = \varkappa = 2$, temos $\kappa_\ell = \{p : (C_0 \to C_1 \to C_1 \to C_0 \to C_1 \to \alpha) \to (C_1 \to C_1 \to C_1 \to C_1 \to \alpha) \to \beta\}$.

Simplificação e Inferência de Tipos

O algoritmo de simplificação apenas implementa o processo descrito na Seção 3.1.2, considerando que a política de sobrecarga não permite instâncias sobrepostas. O algoritmo de inferência de tipos, que chamamos de CT_A, é definido na Figura 3.12, sendo apresentado na forma de um sistema de tipos. As regras desse sistema têm a forma $\Gamma_0 \vdash^A e : (\kappa, \Gamma)$ onde (κ, Γ) é a tipagem principal da expressão e no contexto Γ_0 e $\sigma = gen'(\kappa.\tau, \Gamma)$, sendo que $gen'(\kappa.\tau, \Gamma)$ denota uma generalização de $\kappa.\,\tau$ sobre as variáveis de tipo que ocorrem em $\kappa.\tau$ e não ocorrem em Γ. A notação $\varepsilon(\Gamma, \Gamma')$ é definida por:

$$\varepsilon(\Gamma, \Gamma') = \{\tau = \tau' \mid x : \tau \in \Gamma \text{ e } x : \tau' \in \Gamma'\}$$

3.3 Semântica

Definimos a semântica das expressões apresentadas na Figura 3.1 através da tradução dessas para expressões similares, de uma linguagem que é basicamente o núcleo de *ML*. Essa tradução é feita renomeando os identificadores de funções sobrecarregados para novos nomes ou símbolos que ainda não foram usados. Uma expressão de tipo $\forall\bar{\alpha}.\kappa.\tau$, onde $\kappa = \{o_i : \tau_i\}^{i=1..n}$, é interpretada como uma função com n parâmetros extras, $o_1, ...o_n$, onde cada um corresponde a uma restrição que ocorre em κ. Supomos que a cada conjunto de restrições corresponde uma

$$
\begin{array}{l}
[\![\sigma]\!]_\Gamma = \begin{cases} \forall\alpha_j.\tau_1 \to \ldots \to \tau_n \to \tau & \text{if } \kappa' = \{o_i : \tau_i\}^{i=1..n},\ n > 0 \\ \forall\alpha_j.\,\tau & \text{if } \kappa' = \emptyset \end{cases} \\
\text{onde } \sigma = \forall\alpha_j.\,\kappa.\,\tau,\ \ \Gamma \models \kappa \gg \kappa' \\
\\
[\![\emptyset]\!]_\Gamma \quad\ = \emptyset \\
[\![\Gamma', x : \sigma]\!]_\Gamma \ = ([\![\Gamma']\!]_\Gamma), x : [\![\sigma]\!]_\Gamma
\end{array}
$$

Figura 3.13: Tradução de Tipos e Contextos de Tipos

ordem arbitrária, que define a ordem dos parâmetros. A tradução de contextos de tipos Γ e tipos σ do sistema *CT* para contextos e tipos em *ML* é definida na Figura 3.13. A tradução de expressões válidas no sistema *CT* para expressões de *ML* é apresentada na Figura 3.14. As regras para derivação de expressões válidas nessa figura têm a forma:

$$\Gamma \vdash e : \sigma \rightsquigarrow \mathbf{e}$$

onde $\mathbf{e}$ representa a tradução de e.

Essa tradução é feita substituindo expressões $\texttt{let}_o$ que introduzem funções ou símbolos sobrecarregados por expressões `let`. Quando um símbolo sobrecarregado o é resolvido (em um contexto onde é requerido o como tendo um determinado tipo), um novo identificador associado a implementação correspondente é usado na função, o exemplo abaixo ilustra essa idéia:

$$
\begin{array}{lcl}
\texttt{let}_o\ (\texttt{==}) = \mathit{primEqInt} & & \texttt{let}\ (\texttt{==})_I = \mathit{primEqInt} \\
\texttt{let}_o\ (\texttt{==}) = \mathit{primEqFloat} & & \texttt{let}\ (\texttt{==})_F = \mathit{primEqFloat} \\
\qquad\vdots & \rightsquigarrow & \qquad\vdots \\
\texttt{in}\ (x == 1) \vee (y == 1.0) & & \texttt{in}\ ((\lambda o.o)\ (\texttt{==})_I\ x\ 1) \\
 & & \quad \vee((\lambda o.o)\ (\texttt{==})_F\ y\ 1.0)
\end{array}
$$

primEqInt e *primEqFloat* representam funções primitivas que implementam respectivamente a igualdade entre valores inteiros e de ponto flutuante. $(\texttt{==})_I$ e $(\texttt{==})_F$ são novos identificadores associados a essas implementações que são usados como parâmetros na tradução de uma ocorrência do símbolo sobrecarregado (==). O símbolo (==) é traduzido para a abstração-λ $\lambda o.o$, onde seu parâmetro corresponde a restrição $\{(\texttt{==}) : \alpha \to \alpha \to \mathit{Bool}\}$ imposta ao tipo deste símbolo. A escolha de uma instância que satisfaz a restrição em um determinado contexto é traduzida como a passagem da implementação correspondente como argumento para essa abstração-λ.

$$\Gamma, \{x : \sigma_i\}^{i=1..n} \vdash x : \sigma \rightsquigarrow \mathbf{e} \qquad \text{(VARo)}$$

onde $(\mathbf{e}, \sigma) = \begin{cases} (x, \sigma_1) & \text{if } n = 1 \\ (\lambda x.\, x, \{o : \tau\}.\, \tau) & \text{if } n > 1, \text{ onde } \tau = lcg(\{\sigma_i\}^{i=1..n}) \end{cases}$

$$\frac{\Gamma \vdash e : \forall(\alpha_j)^{j=1..m}.\kappa.\, \tau \rightsquigarrow \mathbf{e} \quad \Gamma \models \kappa' \quad \Gamma \models \kappa' \gg \kappa''}{\Gamma \vdash e : \kappa''.\, (\tau[\tau_j/\alpha_j]^{j=1..m}) \rightsquigarrow \lambda\, \langle \kappa'', \Gamma \rangle.\, \mathbf{e}\ args(\kappa'', \kappa, \Gamma)} \qquad \text{(INSTo)}$$

onde $\kappa' = \kappa[\tau_j/\alpha_j]^{j=1..m}$

$$\frac{\Gamma \vdash e : \sigma \rightsquigarrow \mathbf{e}}{\Gamma \vdash e : \forall\alpha.\, \sigma \rightsquigarrow \mathbf{e}} \quad \alpha \notin tv(\Gamma) \qquad \text{(GENo)}$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \rightsquigarrow \mathbf{e_1} \quad \Gamma, \{o : \sigma_1\} \vdash e_2 : \kappa_2.\, \tau_2 \rightsquigarrow \mathbf{e_2} \quad \Gamma \models \kappa}{\Gamma \vdash let\ o = e_1\ in\ e_2 : \kappa.\, \tau_2 \rightsquigarrow \lambda\langle \kappa, \Gamma \rangle.\, let\ o = \mathbf{e_1}\ \langle \kappa_1, \Gamma \rangle\ in\ \mathbf{e_2}\ \langle \kappa_2, \Gamma \rangle} \qquad \text{(LETo)}$$

onde $\kappa = \kappa_1 \cup \kappa_2$, $\sigma_1 = gen(\kappa_1.\, \tau_1, \Gamma)$

$$\frac{\Gamma, \{u : \tau'\} \vdash e : \kappa.\, \tau \rightsquigarrow \mathbf{e}}{\Gamma \vdash \lambda u.\, e :\ \kappa.\, \tau' \to \tau \rightsquigarrow \lambda\langle \kappa, \Gamma \rangle.\, \lambda u.\, \mathbf{e}\ \langle \kappa, \Gamma \rangle} \qquad \text{(ABSo)}$$

$$\frac{\Gamma \vdash e_1 : \kappa_1.\, \tau_2 \to \tau_1 \rightsquigarrow \mathbf{e_1} \quad \Gamma \vdash e_2 : \kappa_2.\, \tau_2 \rightsquigarrow \mathbf{e_2} \quad \Gamma \models \kappa_1 \cup \kappa_2}{\Gamma \vdash e_1\, e_2 : \kappa_1 \cup \kappa_2.\, \tau_1 \rightsquigarrow \lambda\langle \kappa_1 \cup \kappa_2, \Gamma \rangle.\, (\mathbf{e_1}\ \langle \kappa_1, \Gamma \rangle)\ (\mathbf{e_2}\ \langle \kappa_2, \Gamma \rangle)} \qquad \text{(APPLo)}$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \rightsquigarrow \mathbf{e_1} \quad \Gamma; \{o : (\sigma_1, o_{\sigma_1})\} \vdash p : \kappa_2.\, \tau_2 \rightsquigarrow \mathbf{e_2} \quad \Gamma \models \kappa}{\Gamma \vdash let_o\ o = e_1\ in\ e_2 : \kappa.\, \tau_2 \rightsquigarrow \lambda\langle \kappa, \Gamma \rangle.\, let\ o_{\sigma_1} = \mathbf{e_1}\langle \kappa_1, \Gamma \rangle\ in\ \mathbf{e_2}\ \langle \kappa_2, \Gamma \rangle} \qquad \text{(LETOo)}$$

onde o_{σ_1} é uma variável livre, $\kappa = \kappa_1 \cup \kappa_2, \sigma_1 = gen(\kappa_1.\, \tau_1, \Gamma)$

Figura 3.14: Semântica do sistema CT

$$\lambda\, \langle \kappa, \Gamma \rangle.\, \mathbf{e} = \begin{cases} \lambda\, o_1.\ \ldots\ .\lambda\, o_n.\, \mathbf{e} & \text{if } \kappa' = \{o_i : \tau_i\}^{i=1..n},\ n > 0 \\ \mathbf{e} & \text{if } \kappa' = \emptyset \end{cases}$$
onde $\Gamma \models \kappa \gg \kappa'$

$$\mathbf{e}\ \langle \kappa, \Gamma \rangle = \begin{cases} \mathbf{e}\ o_1\ \ldots\ o_n & \text{if } \kappa' = \{o_i : \tau_i\}^{i=1..n},\ n > 0 \\ \mathbf{e} & \text{if } \kappa' = \emptyset \end{cases}$$
onde $\Gamma \models \kappa \gg \kappa'$

$$\mathbf{e}\ \mathbf{a} = \begin{cases} \mathbf{e}\ \mathbf{e}_1\ \ldots\ \mathbf{e}_n & \text{if } \mathbf{a} = \langle\, \mathbf{e}_i \,\rangle^{i=1..n},\ n > 0 \\ \mathbf{e} & \text{if } \mathbf{a} = \langle\, \rangle \end{cases}$$

Figura 3.15: Notação

$$
\begin{array}{l}
args\,(\kappa_1, \kappa_2, \Gamma) = \langle\, \mathbf{e}_1, \ldots, \mathbf{e}_n \,\rangle \\
\quad \text{onde } \Gamma \models \kappa_1 \gg \kappa \\
\qquad \Gamma \models \kappa_2 \gg \kappa' \\
\qquad \{o_i : \tau_i\}^{i=1..n} = \kappa \\
\qquad \mathbf{e}_i = arg(o_i : \tau_i, \kappa', \Gamma), \text{ for } i = 1..n \\
\\
\quad arg\,(o : \tau, \kappa, \Gamma) = \\
\qquad \textbf{if } o : \tau \in \kappa \textbf{ then } o \textbf{ else } o' \textbf{ a} \\
\qquad\quad \text{onde } (\forall \alpha_j.\, \kappa'.\, \tau', o') \in \Gamma(o) \text{ é tal que } S\tau' = \tau, \text{ para algum } S \\
\qquad\quad \mathbf{a} = args\,(S\kappa', \kappa, \Gamma)
\end{array}
$$

Figura 3.16: Argumentos para a Tradução de Expressões que tem Tipos onde são Aplicadas Restrições

As notações usadas na definição semântica são dadas na Figura 3.15, e a função *args* é definida na Figura 3.16, essa função retorna a lista de argumentos que serão aplicados na tradução de um tipo com restrição.

Se $\Gamma \vdash e : \sigma \rightsquigarrow \mathbf{e}$ *é provável então* $[\![\Gamma]\!]\Gamma \vdash^{DM} \mathbf{e} : [\![\sigma]\!]\Gamma$ *também é provável*. Onde $\Gamma \vdash^{DM} e : \sigma$ representa uma fórmula derivável mediante o uso das regras de derivação do sistema de tipos de Damas-Milner.

3.4 Ambigüidade

Se uma expressão possuir mais de uma derivação de tipo e as semânticas atribuídas a essa expressão, usando cada uma das derivações, forem distintas, diz-se que a semântica não é *coerente* [48, 26]. Vejamos o exemplo abaixo, onde existem duas definições para a função *coerce*.

```
coerce x = if x then 1 else 0
coerce x = if x then 1.0 else 0.0
```

Vamos considerar também que existem definições sobrecarregadas para a função *show*, com tipos *Int*→*String* e *Float*→*String*, e que *show* 0 = “0”, *show* 1 = “1”, *show* 0.0 = “0.0”e *show* 1.0 = “1.0”. Então, existem dois valores (semânticas) possíveis para a expressão *show* (*coerce* `true`): podemos usar a definição de *coerce* que converte um valor do tipo *Bool* em *Int* ou a definição que converte um valor do tipo *Bool* em *Float*, obtendo como respostas “1”ou “1.0”, respectivamente. Note que se a definição da instância sobrecarregada da função *show* para valores

do tipo *Float* fosse tal que *show* 0.0 = "0"e *show* 1.0 = "1", as duas derivações possíveis forneceriam a mesma semântica e, dessa forma, não ocorreria o problema de coerência.

É claro que garantir e provar a equivalência da semântica de todas as traduções possíveis para uma expressão, em linguagens envolvendo recursão e não terminação, é em geral um problema indecidível. A abordagem usual é identificar um conjunto de expressões para o qual a propriedade de coerência pode ser verificada, e rejeitar as demais. Dessa forma, o fato de poder existir mais de uma derivação para uma expressão com um dado tipo e semânticas distintas caracteriza essa expressão como *ambígua*, e faz com que tal expressão seja rejeitada. Isso é o que ocorre no caso de expressões como *show* (*coerce* `true`), no exemplo apresentado acima.

O processo de verificação da satisfazibilidade das restrições de um tipo polimórfico $\sigma = \forall\bar{\alpha}.\kappa.\tau$, onde $\kappa = \{o_i : \tau_i\}^{i=1..n}$, em um contexto de tipos Γ envolve encontrar todas as suposições de tipo de o_i em Γ que satisfazem ao conjunto de restrições $\{o_i : \tau_i\}^{i=1..n}$. A função *sats*, definida na Figura 3.10 retorna um conjunto de substituições $\mathbb{S} = \{R_1, ..., R_n\}$ para o qual cada R_j, $j \in \{1, \ldots, m\}$, deve ser tal que $R_j\{o_i : \tau_i\}^{i=1..n}.\tau$ é uma instância de σ que satisfaz ao conjunto de restrições $\{o_i : \tau_i\}^{i=1..n}$. Um tipo é considerado ambíguo, no sistema CT, se e somente se:

$$\exists i, j \in \{1, \ldots, m\} \text{ tais que } i \neq j,\ R_i\kappa \neq R_j\kappa \text{ e } R_i\tau = R_j\tau \qquad \texttt{(amb-1)}$$

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

Podemos então conjecturar que: *se* $\Gamma \vdash e \rightsquigarrow e_1 : \sigma$, $\Gamma \vdash e \rightsquigarrow e_2 : \sigma$ *são prováveis e* σ *não é ambíguo em* Γ, *então* $e_1 = e_2$. Uma prova formal dessa conjectura está sendo preparada.

A regra `(amb-1)` pode ser relaxada, permitindo que um conjunto maior de expressões sejam consideradas como válidas, se definirmos como ambíguas apenas as expressões que tenham mais de uma tradução em qualquer contexto onde ocorram. Ou seja, uma expressão é considerada ambígua se e somente se:

$$\forall i \in \{1..m\}\ \ \exists j \neq i, j \in \{1..m\}, \text{ tal que } R_i\kappa \neq R_j\kappa \text{ e } R_i\tau = R_j\tau \qquad \texttt{(amb-1')}$$

Adotando essa regra, uma expressão como, por exemplo, $f\ o$, não é tratada como ambígua no contexto $\{f : Int \rightarrow Int, f : Int \rightarrow Float, f : Float \rightarrow$

$$\{o:\tau\}|_V = \textbf{ if } tv(\tau) \cap V = \emptyset \textbf{ then } \emptyset \textbf{ else } \{o:\tau\}$$
$$(\{o:\tau\} \cup \kappa')|_V = \{o:\tau\}|_V \cup \kappa'|_V$$

$$\kappa|_V^* = \begin{cases} \kappa|_V & \text{se } tv(\kappa|_V) \subseteq V \\ \kappa|_{tv(\kappa|_V)}^* & \text{caso contrário} \end{cases}$$

Figura 3.17: Projeção das restrições

$Float, o : Int, o : Float\}$. A motivação para essa regra é que, mesmo existindo duas traduções possíveis para $f\ o$ quando essa expressão tem tipo *Float*, ela pode ser perfeitamente usada em um contexto onde ocorra com tipo *Int*. A expressão somente é rejeitada quando usada em um contexto onde não existe a possibilidade de nenhuma tradução válida. É importante ressaltar que essa nova definição de ambigüidade contraria a definição usual de coerência, sendo necessária uma análise mais detalhada para determinar as vantagens e desvantagens de sua adoção.

Em uma abordagem de *mundo fechado*, a regra `(amb-1)` é suficiente para rejeitar todas as expressões que tenham a possibilidade de violar a coerência da linguagem. Mas, para uma adaptação do sistema *CT* para suporte também a uma abordagem de *mundo aberto*, como discutido na Seção 3.5, uma regra adicional deve ser usada juntamente com `(amb-1)`. Isso ocorre porque, em uma abordagem de mundo aberto, nem todas (ou talvez nenhuma) das instâncias de um determinado tipo precisam ser elementos do contexto de tipos, quando uma expressão desse tipo é usada. Nesse caso, uma expressão é considerada ambígua se satisfaz a regra `(amb-1)` (ou `(amb-1')`) e a regra `(amb-2)` definida como:

$$\kappa \neq \kappa|_{tv(\tau,\Gamma)}^*$$

onde $\kappa|_V^*$ denota a operação de projeção de restrições definida na Figura 3.17. A idéia intuitiva de projeção de restrições é a de extrair a relação entre as variáveis de tipo presentes em τ e as variáveis de tipo presentes em κ, verificando se algumas das restrições definem uma dependência entre as mesmas. Por exemplo, se $\kappa = \{f : \alpha \to \beta, o : \beta\}$ então o tipo $\kappa.\beta$ não é considerado ambíguo, uma vez que $\kappa|_{\{\beta\}}^* = \kappa$.

A regra `(amb-2)` elimina a necessidade de especificação, pelo programador, de dependências funcionais entre as variáveis de tipo. Ela permite, no entanto, o uso de expressões polimórficas que, se utilizadas em qualquer contexto no qual fossem instanciadas, provocariam a detecção de um erro de tipo. Por exemplo:

$$\begin{aligned}\Gamma_* = \{\quad & (*) : Int \to Int \to Int, && (*) : Float \to Float \to Float \\ & (*) : Int \to Float \to Float, && (*) : Float \to Int \to Float \\ & (*) : Int \to Float \to Int, && (*) : Float \to Int \to Int \\ & (*) : Int \to Int \to Float\}\end{aligned}$$

Nesse contexto, o tipo inferido para a expressão (1∗2.5)∗3 é:

$$(1*2.5)*3 : \{(*) : Int \to Float \to \alpha, (*) : \alpha \to Int \to \beta\}.\, \beta$$

Existem quatro substituições que satisfazem esse tipo:

$$\begin{aligned}\{\quad & \{\alpha \mapsto Int, \beta \mapsto Int\}, \{\alpha \mapsto Float, \beta \mapsto Int\}, \\ & \{\alpha \mapsto Int, \beta \mapsto Float\}, \{\alpha \mapsto Float, \beta \mapsto Float\}\}\end{aligned}$$

Como existem duas substituições possíveis para todo contexto em que a sobrecarga poderia ser resolvida (tanto no caso do tipo da expressão ser *Int* quanto no caso de ser *Float*), um erro de tipo poderia ser detectado (e seria, tanto mediante o uso da regra `(amb-1)` quanto da regra `(amb-1')`). Note, no entanto, que, se a expressão acima fosse usada em qualquer contexto com tipo *Int* ou *Float*, a regra `(amb-2)` detectaria um erro de ambigüidade.

3.5 Mundo Fechado *versus* Mundo aberto

O sistema de tipos de *Haskell* adota uma abordagem de mundo aberto, que pode ser caracterizada informalmente pelas seguintes características:

- O tipo principal de todo símbolo sobrecarregado tem que ser explicitamente anotado (em *Haskell*, em declarações de classes).
- Para todo símbolo sobrecarregado, a definição de uma determinada instância desse símbolo deve estar visível em um dado contexto apenas quando a sobrecarga é resolvida.

Na abordagem de mundo fechado, por outro lado, as declarações de um símbolo visíveis em um contexto são usadas para determinar o tipo das expressões que fazem referência a este símbolo (e, é claro, se estas são bem tipadas ou não). Isso é exatamente o que ocorre na tipagem de expressões que não envolvem sobrecarga. Podemos dizer, assim, que a abordagem de mundo fechado é uma extensão natural do sistema de Damas-Milner para tratamento de sobrecarga. A abordagem de mundo fechado permite que o tipo inferido para declarações que façam

uso de símbolos sobrecarregados possam ser "aprimorados"[2], e erros relacionados à satisfazibilidade de restrições possam ser detectados antecipadamente. Em contrapartida, essa abordagem não permite que um símbolo sobrecarregado seja usado, sem que pelo menos duas definições de instâncias desse símbolo tenham sido definidas, de modo a que o símbolo sobrecarregado possa ser usado de forma polimórfica, o que é uma dificuldade significativa na presença de compilação separada de módulos.

Para ilustrar a diferença ente as duas abordagens considere o contexto $\Gamma_g = \{g : \mathit{Bool} \rightarrow \mathit{Char},\ \ g : \mathit{Char} \rightarrow \mathit{Bool}\}$. Na abordagem de mundo fechado, uma declaração como, por exemplo, x `=` $g\ \mathit{True}$ tem seu tipo inferido como Char. Em uma extensão de *Haskell* com múltiplos parâmetros, as declarações sobrecarregadas de g teriam que ser instâncias de uma classe definida como a seguir:

$$\texttt{class}\ G\ a\ b\ \texttt{where}$$
$$g :: a \rightarrow b$$

Nesse caso, x declarado acima tem como tipo principal $(G\ \mathit{Bool}\ \alpha) \Rightarrow \alpha$. Isso ocorre porque, na abordagem de mundo aberto, considera-se que é possível existir uma nova instância para G, em um outro contexto (possivelmente em um novo módulo), com g tendo, por exemplo, tipo $\mathit{Bool} \rightarrow \mathit{Int}$, que possa ser usada quando o tipo de x for instanciado (no exemplo, para o tipo Int).

O sistema CT pode ser estendido para prover suporte também à abordagem de mundo aberto. Uma proposta para suporte à abordagem de mundo aberto é baseada simplesmente no uso de uma cláusula específica para anotação de tipos para os símbolos sobrecarregados, chamada de cláusula `assume`. Essa cláusula especifica apenas tipos simples, sem restrições, para símbolos sobrecarregados (as restrições que serão aplicadas para uma determinada instância dessa declaração serão inferidas de acordo com sua definição). Por exemplo, a declaração que define um tratamento de mundo aberto para o operador de igualdade (`==`) é feita simplesmente como a seguir:

$$\texttt{assume}\ (\texttt{==}) :: a \rightarrow a \rightarrow \mathit{Bool}$$

Em geral, uma anotação de tipos `assume` $x :: \tau$ introduz no contexto de tipos inicial uma *suposição de tipos de mundo aberto* $x : \forall\bar{\alpha}.\{x : \tau\}.\tau$, onde $\bar{\alpha}$ é uma

[2]Em uma extensão de *Haskell* com classes de múltiplos parâmetros o aperfeiçoamento dos tipos também é realizado, mas para isso é necessário que as dependências entre os parâmetros tenham sido especificadas explicitamente pelo programador.

$$\Gamma \vdash x : \sigma \qquad \text{(VARo')}$$

$$\text{onde } \sigma = \begin{cases} \sigma_1 & \text{if } ow(\Gamma)(x) = \sigma_1 \text{ ou } cw(\Gamma)(x) = \sigma_1 \\ \{x : \tau\}.\tau & \text{if } cw(\Gamma)(x) = \{\sigma_i\}^{i=1..n}, \text{onde } \tau = lcg(\{\sigma_i\}^{i=1..n}) \end{cases}$$

$$\frac{\Gamma \vdash e : \forall(\alpha_j)^{j=1..m}.\kappa.\tau \quad cw(\Gamma) \vDash cw(\kappa') \quad cw(\Gamma) \vDash cw\,\kappa' \gg \kappa''}{\Gamma \vdash e : \kappa''.\tau'} \qquad \text{(INSTo')}$$

$$\text{onde } \kappa'.\tau' = (\kappa.\tau)[\tau_j/\kappa_j]^{j=1..m}$$

Figura 3.18: Alteração no sistema CT para suporte a abordagem de mundo aberto

seqüência de variáveis formada pelas variáveis que ocorrem em τ. As suposições de mundo aberto em um contexto de tipos recebem um tratamento especial, como explicado a seguir. Seja $ow(\Gamma)$ o conjunto de suposições de mundo aberto contidas no contexto de tipos Γ, e $cw(\Gamma) = \Gamma - ow(\Gamma)$ o seu complemento — i.e. as demais suposições de tipo em Γ, que também chamamos agora de *suposições de tipo de mundo fechado*. Para um conjunto de restrições são definidos os seguintes predicados:

$$\begin{aligned} cw(k) = \{o : \tau \in \kappa \mid\ & tv(\tau) = \emptyset \text{ ou } o : \tau \text{ é uma restrição originada} \\ & \text{de uma suposição de mundo fechado } \} \\ ow(\kappa) = \kappa - cw(\kappa) & \end{aligned}$$

Também são acrescentadas duas novas condições às definidas na Seção 3.1.1 para caracterização de um contexto válido:

- $ow(\Gamma)$ contém apenas uma suposição de tipo para cada símbolo.
- $x : \tau \in ow(\Gamma)$ e $x : \kappa.\tau' \in cw(\Gamma)$ implica em $inst(\tau', \tau)$.

Para o sistema de tipos prover suporte às duas abordagens, são necessárias pequenas modificações, apenas nas regras `(VARo)` e `(INSTo)`. Essas mudanças estão destacadas na Figura 3.18. Essas alterações se refletem na regra `(VARo)` do algoritmo de inferência de tipos, como destacado na Figura 3.19. No algoritmo também são necessárias as substituições das funções *sat* e *simplificar* por *sat'* e *simplificar'*, respectivamente, que verificam a satisfazibilidade das restrições e executam a simplificação apenas para as restrições de mundo fechado ou que não envolvem variáveis de tipo. As funções *sat'* e *simplificar'* são definidas como:

$$\Gamma_0 \vdash^A x : \kappa.\tau \qquad \text{(VARo')}$$

$$\text{onde } \kappa.\tau = \begin{cases} \alpha & \text{if } \Gamma_0(x) = \emptyset, \text{ onde } \alpha \text{ é uma var. de tipo livre} \\ \kappa'\tau' & \text{if } ow(\Gamma_0)(x) = \forall\bar{\alpha}.\kappa'.\tau' \text{ ou } cw(\Gamma_0)(x) = \forall\bar{\alpha}.\kappa'.\tau', \\ & \text{onde} \\ & \kappa' = simplificar(\kappa[\bar{\beta}/\bar{\alpha}], \Gamma_0) \\ & \tau' = \tau[\bar{\beta}/\bar{\alpha}], \text{ onde } \bar{\beta} \text{ são vars. de tipos livres} \\ \{x : \tau''\}.\tau'' & \text{if } cw(\Gamma_0)(x) = \{\sigma_i\}^{i=1..n} \text{ e } n > 1 \\ & \text{onde } \tau'' = lcg(\{\sigma_i\}^{i=1..n}) \end{cases}$$

Figura 3.19: Algoritmo para Inferência Alterado para Suporte a Mundo Aberto em CT

$$\begin{aligned} sat'(\kappa, \Gamma) &= sat(cw(\kappa), cw(\Gamma)) \\ simplificar'(\kappa, \Gamma) &= ow(\kappa) \cup simplificar(cw(\kappa), cw(\Gamma)) \end{aligned}$$

3.6 Implementação de inferência de tipos em uma linguagem baseada no sistema CT

O sistema de tipos definido nesse capítulo não trata definições recursivas, ou seja, não é incluída nenhuma regra para introdução de definições usando o operador de ponto fixo, assim como não é incluída nenhuma regra para introdução de definições mutuamente recursivas em expressões `let` (ou `letrec`). A implementação do algoritmo de inferência de tipos descrita nesse trabalho é baseada em uma extensão de trabalhos anteriores (veja [55, 18]), para suporte a sobrecarga. Apesar de não termos ainda definido um sistema de tipos que trate tanto definições sobrecarregadas quanto definições mutuamente recursivas, a experiência com a implementação desse algoritmo tem demonstrado não existir nenhum problema em se trabalhar em conjunto com essas duas formas de declaração. No próximo capítulo descrevemos essa implementação e apresentamos os resultados obtidos.

Como discutido na Seção 2.2.4, em linguagens que adotam o sistema Damas-Milner ou alguma de suas extensões, a abordagem mais usual para tratamento de definições mutuamente recursivas não permite o uso polimórfico de uma função (ou símbolo) no contexto onde ela foi declarada. O *front-end* do compilador realiza uma ordenação topológica das declarações que ocorrem em um mesmo

escopo, de forma a realizar a inferência de tipos de declarações antes de outras que usam nomes introduzidos nas primeiras. No caso de declarações mutuamente recursivas, e que portanto não poderiam ser linearmente ordenadas, seus tipos são considerados como monomórficos. A implementação descrita nesse trabalho adota uma abordagem mais flexível, que permite a inferência de tipos polimórficos em um mesmo contexto, eliminando a necessidade da ordenação topológica.

4 Implementação

Nesse capítulo discutimos a implementação e os primeiros resultados obtidos com o protótipo de um *front-end* para um interpretador da linguagem *Haskell*, ligeiramente modificada para utilizar o sistema de tipos CT. Como nesse sistema não existe a necessidade de declarações adicionais para o tratamento de sobrecarga, todas as construções sintáticas relativas a declarações de classes e instâncias foram eliminadas da linguagem.

O analisador sintático foi implementado usando a biblioteca de combinadores monádicos *Parsec* [36]. Como trata-se de um protótipo, nem todas as abreviações sintáticas (*syntax sugar*) presentes na gramática de *Haskell* foram incluídas. Por exemplo, a notação *do*, que fornece uma sintaxe mais convencional para expressões envolvendo mônadas.

A implementação da inferência de tipos é baseada no algoritmo de Mark Jones para a linguagem *Haskell* [28], mas explora a idéia, sugerida por Trevor Jim [24], de usar tipagem principal para tratar declarações mutuamente recursivas. A verificação da satisfazibilidade das restrições aplicadas ao tipo inferido é executada em um processo separado, que ocorre após terem sido determinados os tipos principais de todas as declarações. Caso todas as restrições de tipo sejam satisfeitas, as restrições sobre os tipos inferidos são simplificadas, sempre que possível.

A seqüência de passos executadas pelo protótipo pode ser resumida como:

1. Análise sintática do programa.
2. Cálculo da *tipagem principal inicial* de cada uma das declarações.
3. Semi-unificação dos tipos inferidos e requeridos.
4. Remoção das restrições relativas à própria declaração.
5. Remoção de restrições que envolvam símbolos que não são sobrecarregados, substituindo-as pelas restrições presentes em sua suposição de tipo.
6. Verificação da existência de suposições de tipos sobrepostas, o que caracterizaria o contexto como inválido de acordo com a política de sobrecarga.

7. Verificação da satisfazibilidade das restrições.

8. Simplificação das restrições.

4.1 Declarações Mutuamente Recursivas

A primeira etapa da inferência de tipos tem como entrada um conjunto de declarações $G = \{x_i : e_i\}^{i=1..n}$ e um contexto inicial Γ sendo que, ao invés de calcular apenas o "tipo principal inicial" de cada uma das expressões e_i, o algoritmo calcula sua "tipagem principal inicial", que é representada pelo par $(\kappa_i.\tau_i, \Gamma_i)$, onde $\kappa_i.\tau_i$ representa o tipo principal inicial da expressão e_i e Γ_i o contexto com as suposições de tipos mínimas (estritamente necessárias) para os símbolos que ocorrem livres em e_i[1]. São permitidas mais de uma suposição de tipo para cada símbolo no contexto Γ_i, e cada uso de um símbolo livre gera uma nova suposição de tipo neste contexto. Chamamos o par $(\kappa_i.\tau_i, \Gamma_i)$ tipagem principal inicial porque ele não corresponde exatamente à tipagem principal para a expressão e_i, uma vez que no momento da inferência não estão disponíveis as suposições de tipo para os símbolos x_i declarados em G. O tipo principal de uma expressão é determinado apenas após o aperfeiçoamento de tipos resultante do processo de verificação da satisfazibilidade.

O conjunto de restrições de um tipo $\kappa.\tau$, inferido para uma expressão e em um contexto inicial Γ nessa primeira etapa, é determinado da seguinte forma. Para cada símbolo x que ocorre em e:

- Se x é um símbolo sobrecarregado em Γ (símbolos que possuem mais de uma suposição de tipo), é incluída em κ a restrição de tipo $x : \tau_g$, onde τ_g é a generalização mínima dos tipos de x em Γ.
- Se x não é um símbolo sobrecarregado em Γ e $(x : \kappa_x.\tau_x) \in \Gamma$, cada restrição que ocorre em κ_x é incluída em κ.
- Se x é um símbolo livre é incluída a restrição $x : \tau_m$, onde τ_m é a suposição de tipo mínima para x em e.

A suposição mínima para um símbolo livre x é adicionada à restrição por não ser possível determinar nesse momento se o símbolo é sobrecarregado, ou ainda se seu tipo possui alguma restrição.

Depois de a tipagem principal inicial para todas as declarações ser obtidas, os tipos inferidos (tipo principal inicial das declarações) e os tipos requeridos

[1]Consideramos livres os símbolos x que ocorrem em e_i tal que $(x : e) \in G$. Esse símbolos são considerados livres porque ainda não têm uma suposição de tipo associada a eles.

(suposições mínimas para os símbolos que ocorrem livres) são semi-unificados. A substituição resultante desse processo é então aplicada aos tipos inferidos.

O algoritmo que infere a tipagem principal inicial de uma expressão é apresentado na Figura 4.1. Ele tem como entrada uma expressão e e o contexto inicial Γ produzindo como resultado uma tripla $(\kappa.\tau, \Gamma', \Gamma_{let})$, onde $(\kappa.\tau, \Gamma')$ representam a tipagem principal da expressão e e o contexto Γ_{let} contém as suposições de tipo para as variáveis definidas através de declarações `let` em e. O contexto Γ_{let} é necessário porque uma declaração feita em um escopo mais externo (e que não teve ainda seu tipo determinado) pode ser usada na definição de um símbolo dentro de uma expressão `let`. O processo de semi-unificação dos tipos inferidos e requeridos pode ocorrer apenas quando todas as declarações estiverem disponíveis, e dessa forma as suposições de tipos inferidos, para cada um dos símbolos definidos através de declarações `let` em uma expressão e, devem ser retornadas em Γ_{let}, para posteriormente serem semi-unificadas em conjunto com todas as outras declarações.

O primeiro passo desse processo de semi-unificação é a criação de um conjunto de pares contendo os tipos inferidos e requeridos para cada um dos símbolos declarados em G, sendo cada elemento composto de um tipo requerido para um determinado símbolo x e a suposição de tipo inferida para este símbolo. Caso exista mais de uma declaração correspondente ao símbolo (o que significa que se trata de um símbolo sobrecarregado) é usada a generalização mínima das suposições inferidas que unificam com o tipo requerido. Cada um desse pares corresponde a uma inequação $(\tau_{inf\,j} \leq^j \tau_{req\,j})^{j=1..m}$ que compõem uma instância do problema de semi-unificação, onde Γ_i é o conjunto dos tipos requeridos para cada expressão e_i, e m é o número de suposições em $\bigcup^{i=1..n} \Gamma_i$. Esse processo é definido na Figura 4.2.

A semi-unificação é resolvida através de uma seqüência de substituições das variáveis que ocorrem nos tipos inferidos por variáveis livres (um processo que chamamos renomeação) e unificações destes com os tipo requeridos correspondentes. Essa seqüencia é repetida enquanto algum tipo, inferido ou requerido, for atualizado pela aplicação da substituição obtida. O algoritmo retorna um erro quando algum dos pares não puder ser unificado ou uma dependência circular entre as variáveis de tipo que estão sendo unificadas for detectada. Consideramos que uma variável de tipo α apresenta uma dependência circular com uma variável β quando a variável β foi gerada como uma variável livre para substituir α e a substituição resultante da unificação entre inferidos e requeridos implica em trocar β por um tipo em que ocorra a variável α. Essa verificação pode ser implementada fazendo com que as variáveis livres se "lembrem"de todas as variáveis das quais ela se originou, o que pode ser feito através de uma lista "antepassados"associado

a cada variável. Como resultado desse algoritmo é obtido uma substituição que é aplicada ao contexto inicial de tipos inferidos, terminando então o processo de inferência. O algoritmo de semi-unificação apresentado aqui é uma adaptação para suporte a sobrecarga de símbolos do algoritmo apresentado em [55]. A partir desse momento as referências ao processo de semi-unificação dos tipos inferidos e requeridos devem ser entendidas como a execução desse algoritmo.

Em uma etapa posterior, a satisfazibilidade, de cada uma das suposições que ocorrem em tipos de variáveis definidas nesse contexto deve ser verificada, mas antes as restrições referentes a símbolos que não estão sobrecarregados devem ser removidas. Como comentado anteriormente, no momento da inferência do tipo de uma expressão não existe como verificar se um determinado símbolo livre é sobrecarregado ou não, e por isso este símbolo é incluído no conjunto de restrições. Quando se remove uma restrição referente a um símbolo não sobrecarregado essa restrição deve ser substituída pelas restrições aplicadas a este símbolo (realizando as unificações correspondentes). Restrições que envolvam o proprio símbolo também devem ser removidas. Esses processos são relativamente simples e diretos e por isso os algoritmos não são apresentados aqui. Na próxima Seção são apresentados alguns exemplos que ilustram a inferência de tipos.

4.1.1 Exemplos

Considere o contexto de tipos Γ_{eo}, a seguir, e a inferência de tipo para as definições mutuamente recursivas das funções *even* e *odd* abaixo, no contexto Γ_{eo}:

$$\begin{array}{lll} \Gamma_{eo} = \{ & coerce : Int \to Int, & coerce : Int \to Float, \\ & (\texttt{==}) : Int \to Int \to Bool, & (\texttt{==}) : Float \to Float \to Bool, \\ & (\texttt{-}) : Int \to Int \to Bool, & (\texttt{-}) : Float \to Float \to Bool\} \end{array}$$

$$\begin{array}{l} even\ \ n\ \texttt{=}\ \texttt{if}\ \ n == (coerce\ 0)\ \ \texttt{then}\ \ True\ \ \texttt{else}\ \ odd\ \ (n - (coerce\ 1)) \\ odd\ \ n\ \ \texttt{=}\ \texttt{if}\ \ n == (coerce\ 0)\ \ \texttt{then}\ \ False\ \ \texttt{else}\ \ even\ \ (n - (coerce\ 1)) \end{array}$$

O resultado da inferência de tipos para essas definições, ou seja, a tipagem principal inicial para cada uma delas, é mostrada a seguir:

$$\begin{array}{l} even : (\{(\texttt{==}) : \alpha \to \alpha \to Bool, (\texttt{-}) : \alpha \to \alpha \to \alpha, coerce : Int \to \alpha, \\ \qquad odd : \alpha \to Bool)\}\ .\ \alpha \to Bool,\ \{odd : \alpha \to Bool\}) \\ odd : (\{(\texttt{==}) : \beta \to \beta \to Bool, (\texttt{-}) : \beta \to \beta \to \beta, coerce : Int \to \beta, \\ \qquad even : \beta \to Bool)\}\ .\ \beta \to Bool,\ \{even : \beta \to Bool\}) \end{array}$$

$CTA(\Gamma,\ x)$
$\quad$ `=` `if` $\Gamma_{(x)} = \{\forall\bar{\alpha}_i.\kappa_i.\tau_i\}^{i=1..n}$ `then`
$\quad\quad$ `if` $i = 1$ `then` $(\kappa_1.\tau_1, \emptyset, \emptyset)$
$\quad\quad$ `else let` $\tau = lcg(\{\tau_i\}^{i=1..m})$ `in` $(\{x : \tau\}.\tau, \emptyset, \emptyset)$
$\quad$ `else` $(\{x : \alpha\}.\alpha, \{x : \alpha\}, \emptyset)$
$\quad\quad$ onde α é uma variável livre

$CTA(\Gamma,\ e\ e')$
$\quad$ `=` `let` $(\kappa.\tau, \Gamma_e, \Gamma_{let}) = CTA(\Gamma, e)$
$\quad\quad$ $(\kappa'.\tau', \Gamma'_e, \Gamma'_{let}) = CTA(\Gamma, e')$
$\quad\quad$ $S = unificar(\{(\beta, \gamma) \mid$ onde x é uma variável ligada a uma abstração-λ e $x : \beta \in \Gamma_e$ e $x : \gamma \in \Gamma'_e\} \cup \{(\tau, \tau' \to \alpha)\})$ e α é uma variável livre
$\quad$ `in` $((S\kappa \cup S\kappa').S\alpha, S\Gamma_e \cup S\Gamma'_e, \Gamma_{let} \cup \Gamma'_{let})$

$CTA(\Gamma,\ \lambda x.e)$
$\quad$ `=` `let` $(\kappa.\tau, \Gamma', \Gamma_{let}) = CTA(\Gamma - \Gamma_{(x)} \cup \{x : \alpha\}, e)$
$\quad$ `in` $(\alpha \to \tau, \Gamma' - \{x : \alpha\})$
$\quad\quad$ onde α é uma variável livre

$CTA(\Gamma,$ `let` $\{x_i = e_i\}^{i=1..n}$ `in` $e)$ `=`
$\quad$ `let` $(\kappa.\tau, \Gamma', \Gamma_{let}) = CTA(\Gamma, e)$
$\quad\quad$ para $i = 1...n$
$\quad\quad\quad$ $(\kappa_i.\tau_i, \Gamma'_i, \Gamma_{let_i}) = CTA(\Gamma, e_i)$
$\quad\quad\quad$ $S_i = unificar(\{(\beta, \gamma) \mid x : \beta \in \Gamma_{let_i}$ e $x : \gamma \in \Gamma'$ e x é uma variável ligada a uma abstração-$\lambda\}$
$\quad\quad$ $\Gamma'_{let} = \bigcup_{i=1..n} \{x_i : \kappa_i.\tau_i\}$
$\quad\quad$ $\Gamma''_{let} = \bigcup_{i=1..n} \Gamma_{let_i}$
$\quad\quad$ para $j = 1...m$ onde m é o número de elementos $x_j : \tau_j \in \Gamma'$
$\quad\quad\quad$ $\tau' =$ o tipo τ_j com todas suas variáveis de tipo que não são ligadas a uma abstração-λ substituídas por var. livres.
$\quad\quad\quad$ $\tau'' = \Gamma'_{let(x_j)}$
$\quad\quad$ $S'_j = unificar(\tau', \tau'')$
$\quad\quad$ $S = S'_1 \circ S'_2 \circ ... \circ S'_m \circ S_1 \circ S_2 \circ ... \circ S_n$
$\quad$ `in` $(S\kappa.\tau, S\Gamma', S\Gamma_{let} \cup S\Gamma'_{let} \cup S\Gamma''_{let})$

Figura 4.1: Tipagem Principal de uma Expressão

$\Gamma_f = S\Gamma_{inf}$
onde

para $i = 1..n$ onde n é o número de declarações no grupo $\{x_i : e_i\}$

$\texttt{let}\ (\kappa_i.\tau_i, \Gamma_i, \Gamma_{let_i}) = CTA(\Gamma, e_i)$

$\texttt{in}\ \Gamma_{inf} = \bigcup_{j=1..n} (\{x_j : \kappa_j.\tau_j\} \cup \Gamma_{let_j})$

$\Gamma_{req} = \bigcup_{j=1..n} \Gamma_j$

para $i = 1..m$ onde m é o número de elementos em Γ_{req}

$x_i : \tau'_i \in \Gamma_{req}$

$\{\tau_j\}^{j=1..k} = \{\tau \mid \tau = \Gamma_{inf(x_i)}\ \text{e}\ unificar(\tau, \tau'_i)\}$

$$\tau''_i = \begin{cases} Falha & \text{se } k = 0 \\ \tau_1 & \text{se } k = 1 \\ lcg(\{\tau_j\}) & \text{se } k > 1 \end{cases}$$

$S = SU(\{\tau'_i, \tau''_i\}^{i=1..m})$

$SU(\theta) = \texttt{let}\ S = SU'(\theta)$
$\texttt{in if}\ circular\ S\ \texttt{then}\ Falha$
$\texttt{else if}\ dom(S) \cap tv(\theta) = \emptyset$ ou
existir apenas renomeações em S $\texttt{then}\ S \circ SU(\theta)$
$\texttt{else}\ S$

$SU'(\emptyset) = id$
$SU'(\{(\tau, \tau')\} \cup \theta)$ $\texttt{= let}\ S = unificar(\tau_l, \tau')$
$\texttt{in}\ S \circ SU'(\theta)$
sendo τ_l o tipo τ com todas as suas variáveis
de tipos substituídas por varáveis livres

Γ é o contexto de tipos inicial

Figura 4.2: Inferência de Tipos para um Grupo de Declarações

Os tipos atribuídos aos símbolos (==), (-) e à função *coerce*, que aparecem nas restrições impostas aos tipos das funções *even* e *odd*, são obtidos por meio da generalização mínima dos tipos das definições sobrecarregadas para estes símbolos, dados no contexto Γ_{eo}.

A suposição de tipo para *odd*, que aparece na restrição aplicada ao tipo de *even* e no seu contexto de tipagem, é o tipo mínimo requerido para que essa declaração tenha um tipo válido. Como *odd* ocorre livre no contexto em que o tipo de *even* é inferido, essa suposição é incluída tanto na restrição (não é possível determinar nesse momento se este símbolo está sobrecarregado), quanto no contexto retornado como parte da tipagem principal de *even*. Como as duas definições ocorrem em um mesmo escopo, a inferência de tipos para a definição de *odd* é feita de forma similar. Temos, portanto, o seguinte conjunto de pares inferidos e requeridos:

$$\{(\alpha \to Bool, \beta \to Bool), (\beta \to Bool, \alpha \to Bool)\}$$

No passo seguinte do processo de inferência de tipos, ou seja, a unificação de tipos inferidos e requeridos, as variáveis dos tipos inferidos são primeiramente substituídas por variáveis novas:

$$\{(\alpha^1 \to Bool, \beta \to Bool), (\beta^2 \to Bool, \alpha \to Bool)\}$$

A unificação desses tipos produz como resultado a substituição:

$$S = \{(\alpha^1 \mapsto \beta), (\beta^2 \mapsto \alpha)\}$$

Como a aplicação dessa substituição resulta apenas em renomeações de variáveis dos tipos envolvidos (i.e., substituição de uma variável de tipo por outra variável de tipo), a seqüência de iterações de unificação usada no processo de semi-unificação é interrompida. A substituição S obtida é aplicada nos tipos inferidos originalmente, sem resultar em nenhuma alteração dos mesmos.

Considere agora que a definição da função *even* é modificada como mostrado a seguir, de maneira a operar apenas sobre valores de tipo *Int*:

$$\begin{aligned} &even\ n\ \texttt{=}\ \texttt{if}\ n == 0\ \texttt{then}\ True\ \texttt{else}\ odd\ (n-1) \\ &odd\ n\ \ \texttt{=}\ \texttt{if}\ n == (coerce\ 0)\ \texttt{then}\ False\ \texttt{else}\ even\ (n-(coerce\ 1)) \end{aligned}$$

A modificação introduzida na definição de *even* altera o tipo inferido para *odd*, uma vez que *even* é usado na sua definição. As tipagens principais inferidas para essas duas definições são:

$$\begin{aligned} even : & (\{(\texttt{==}) : Int \to Int \to Bool, (\texttt{-}) : Int \to Int \to Int, \\ & \quad odd : Int \to Bool)\} \,.\, Int \to Bool,\ \{odd : Int \to Bool\}) \\ odd : & \ (\{(\texttt{==}) : \alpha \to \alpha \to Bool, (\texttt{-}) : \alpha \to \alpha \to \alpha, coerce : Int \to \alpha, \\ & \quad even : \alpha \to Bool)\} \,.\, \alpha \to Bool,\ \{even : \alpha \to Bool\}) \end{aligned}$$

Os pares de tipos (inferido,requerido) para os símbolos *even* e *odd* são:

$$\{(Int \to Bool, \alpha \to Bool), (\alpha \to Bool, Int \to Bool)\}$$

Renomeando as variáveis desses tipos para variáveis novas, obtemos:

$$\{(Int \to Bool, \alpha \to Bool), (\alpha^2 \to Bool, Int \to Bool)\}$$

A unificação destes produz como resultado a substituição: $S' = \{(\alpha \mapsto Int), (\alpha^2 \mapsto Int)\}$, e ocasiona uma nova iteração do algoritmo, para o conjunto de pares:

$$\{(Int \to Bool, Int \to Bool), (Int \to Bool, Int \to Bool)\}$$

Como a unificação desses pares de tipos resulta na substituição identidade, o processo de semi-unificação termina, retornando a substituição S', que é então aplicada aos tipos inferidos. Após a aplicação dessa substituição, são obtidos os tipos:

$$\begin{aligned} even : & \{(\texttt{==}) : Int \to Int \to Bool, (\texttt{-}) : Int \to Int \to Int, \\ & \quad odd : Int \to Bool)\} \,.\, Int \to Bool \\ odd : & \ \{(\texttt{==}) : Int \to Int \to Bool, (\texttt{-}) : Int \to Int \to Int, coerce : Int \to Int, \\ & \quad even : Int \to Bool)\} \,.\, Int \to Bool \end{aligned}$$

Na última fase da inferência de tipos, as restrições de tipo relativas às funções *even* e *odd* são removidas (uma vez que não são símbolos sobrecarregados) e é verificada a satisfazibilidade das restrições relativas aos símbolos (-) e (==).

A função *coerce* usada nesse exemplo implementa a coerção das constantes inteiras 0 e 1, que podem ter tipos *Int* ou *Float*, dependendo do contexto onde as funções *even* e *odd* forem utilizadas. Optou-se por declarar a coerção de forma explicita apenas para manter coerência com outros exemplos apresentados nesse trabalho, nos quais as constantes 1 e 0 foram tratadas como tendo o tipo *Int*. É claro que a chamada da função *coerce* poderia ser gerada implicitamente pelo

analisador sintático ou, alternativamente, as constantes 0, 1, etc poderiam ser tratadas como sobrecarregadas para valores *Int* e *Float*, uma vez que o sistema *CT* permite a sobrecarga de constantes. Entretanto, essas últimas opções têm como efeito tornar ambíguas expressões tais como, por exemplo, *show 1*.[2]

Além de tratar definições mutuamente recursivas, a inferência de tipos trata também recursão polimórfica, tais como a definição da função *length* sobre árvores perfeitamente balanceadas, apresentada na Seção 2.2.4. A tipagem principal obtida para essa definição é:

$$length : (\{(+) : Int \to Int \to Int, length : Seq\ (\alpha, \alpha) \to Int\}.Seq\ \alpha \to Int,\\ \{(+) : Int \to Int \to Int, length : Seq\ (\alpha, \alpha) \to Int\})$$

O conjunto de pares inferidos e requeridos, após a renomeação para variáveis novas é, portanto: $\{(Seq\ \alpha^1 \to Int, Seq\ (\alpha, \alpha) \to Int)\}$. A unificação realizada no processo de semi-unificação apenas resulta na substituição da variável α^1 por α, o que não altera os tipos inferidos originalmente. O tipo principal de *length* é então determinado como sendo: $\{(+) : Int \to Int \to Int, length : Seq\ (\alpha, \alpha) \to Int\}.Seq\ \alpha \to Int$. A restrição $length : Seq\ (\alpha, \alpha) \to Int$ é então removida, uma vez que *length* não é um símbolo sobrecarregado. Após realizada a verificação da satisfazibilidade de restrições, a restrição $(+) : Int \to Int \to Int$ será removida do tipo inferido para *length*, no processo de simplificação de restrições, já que envolve apenas tipos concretos.

Um caso simples em que é detectada dependência circular entre as variáveis de tipo novas criadas durante o processo de semi-unificação é ilustrado a seguir. Considere a definição:

$$f\ (x, y) = (f\ x, f\ y)$$

A tipagem principal para essa definição é:

$$((a, b) \to (c, d), \{f : a \to c, f : b \to d\})$$

O o conjunto de pares de tipos (inferido, requerido) é, portanto:

$$\{((a^1, b^1) \to (c^1, d^1), a \to c), ((a^2, b^2) \to (c^2, d^2), b \to d)\}$$

[2]Em *Haskell*, esse problema é contornado através de declarações *default*, que permitem especificar uma lista ordenada de tipos que cada constante numérica pode assumir, sendo a ambigüidade resolvida de acordo com a ordem especificada para os tipos nessa lista. Uma solução similar pode ser facilmente adotada em nosso protótipo.

.

A unificação desses pares de tipos produz como resultado a substituição $\{a \mapsto (a^1, b^1), c \mapsto (c^1, d^1), b \mapsto (a^2, b^2), d \mapsto (c^2, d^2)\}$. Nesse caso, existe dependência circular, pois a substituição obtida indica que cada variável deve ser substituída por um tipo no qual ocorre uma variável que tem como origem a própria variável a ser substituída. Essa situação é reportada como erro de tipo. Note que, nesse caso, o processo não terminaria, se não fosse utilizado o teste de dependência circular.

O exemplo a seguir ilustra uma situação um pouco mais complexa, em que ocorre uma dependência circular indireta (causando também um erro na inferência de tipos). Esse exemplo é construído com base nas expressões utilizadas por Henglein [21] na prova de redutibilidade do problema de inferência de tipos para o sistema Milner-Mycroft para o problema de semi-unificação.

Suponha que $(v_1, ..., v_n).\mathbf{i}$, $i = 1..n$, denotam funções de projeção (fst, snd, etc) para n-tuplas, e considere as definições:

$$
\begin{array}{l}
k\ x\ y\ \texttt{=}\ x \\
f\ x_1\ x_2\ \ldots\ x_n\ \texttt{=}\ k\ (\backslash x\ \texttt{->}\ x\ x_1\ x_1\texttt{,}\ \ldots\texttt{,}\ \backslash x\ \texttt{->}\ x\ x_n\ x_n) \\
\qquad (\backslash\ y_1\ y_2\ \cdots\ y_n\ \texttt{->}\ (f\ y_1\ y_2\ \ldots\ y_n)\texttt{.}\mathbf{1}\qquad\ \texttt{==}\ x_2\texttt{,}\ \ldots\texttt{,} \\
\qquad\ \backslash\ y_1\ y_2\ \cdots\ y_n\ \texttt{->}\ (f\ y_1\ y_2\ \ldots\ y_n)\texttt{.}(\mathbf{n-1})\ \texttt{==}\ x_n\texttt{,} \\
\qquad\ \backslash\ y_1\ y_2\ \cdots\ y_n\ \texttt{->}\ (f\ y_1\ y_2\ \ldots\ y_n)\texttt{.}\mathbf{n}\qquad\ \texttt{==}\ x_1\texttt{)}
\end{array}
$$

A complexidade de tempo para detectar a dependência circular dessa declaração cresce exponencialmente com n. Um exemplo similar, também com comportamento exponencial, mas de uma definição bem tipada, pode ser obtido alterando o número de parâmetros da função f para $n+1$ e substituindo x_1 por x_{n+1} na última linha da definição de f. O tipo inferido para a definição modificada dessa forma, para $n = 2$ (exemplo onde f teria 3 parâmetros), é:

$$
\begin{array}{l}
\forall a, b, c, d, e, f, g, h.\ \ a \to ((b \to b \to c) \to c) \to \\
\quad ((((d \to d \to e) \to e) \to ((d \to d \to e) \to e) \to f) \to f) \to \\
\quad ((a \to a \to g) \to g, (((b \to b \to c) \to c) \to ((b \to b \to c) \to c) \to h) \to h)
\end{array}
$$

Esse comportamento exponencial em expressões que apresentam recursão polimórfica, assim como o comportamento exponencial do algoritmo de inferência para o sistema Damas-Milner, ocorre quando o número de variáveis de tipo cresce exponencialmente com tamanho da expressão (*big types*). O algoritmo de inferência de tipos para o sistema de Damas-Milner comporta-se, entretanto, de maneira bastante eficiente na prática, uma vez que expressões com essa característica não ocorrem na prática.

4.2 Verificação da Satisfazibilidade

A verificação de satisfazibilidade de restrições implementada no protótipo corresponde exatamente ao algoritmo apresentado na Seção 3.1.1. A verificação de satisfazibilidade é feita para cada uma das suposições de tipo resultantes do passo anterior do processo de inferência. Um limite de iterações configurável pelo usuário garante a parada do processo para restrições que não possam ser verificadas, uma vez que o problema é indecidível. Mas assim como em sistemas que adotam políticas de sobrecarga bem mais restritivas que definida para o sistema CT, mesmo no caso de um conjunto de restrições que é satisfeito, a verificação tem complexidade exponencial. Os piores casos ocorrem na verificação de satisfazibilidade de restrições em tipos que não apresentam variáveis de tipo em comum, como ilustrado nas Figuras 4.3 e 4.4.

Em cada uma dessas figuras, é mostrada a árvore de chamadas recursivas para a função *sats*. O primeiro exemplo mostra a verificação de satisfazibilidade do conjunto de restrições:

$$\kappa = \{(\texttt{==}) : \alpha \rightarrow \alpha \rightarrow \mathit{Bool}, \mathit{show} : \beta \rightarrow \mathit{String}\}$$

No qual as restrições não possuem variáveis de tipo em comum. Para efeito de comparação, o segundo exemplo ilustra a verificação de satisfazibilidade dessas mesmas restrições, exceto que a variável que ocorre nas duas restrições é agora a mesma, isto é, para a restrição:

$$\kappa = \{(\texttt{==}) : \alpha \rightarrow \alpha \rightarrow \mathit{Bool}, \mathit{show} : \alpha \rightarrow \mathit{String}\}$$

Não é difícil verificar que, para um conjunto de restrições $\{o_i : \tau_i\}^{i=1..n}$, o tamanho do conjunto de suposições que satisfazem às restrições ($cSat$) é, no pior caso (as restrições não possuem variáveis em comum), $t_1 \times ... \times t_n$, onde t_i é o número de suposições que satisfazem a restrição $o_i : \tau_i$. A ocorrência da mesma variável de tipo em mais de uma restrição de tipo reduz o tamanho de $cSat$ e, conseqüentemente, diminui o número de chamadas recursivas à função *sats*. Esse comportamento pode tornar a verificação da satisfazibilidade intratável um situações que ocorrem freqüentemente no desenvolvimento de programas. Considere o seguinte exemplo:

$$\begin{array}{lll} (\texttt{==}) & = & \mathit{primEqInt} \\ (x : xs) \texttt{ == } (y : ys) & = & x == y \ \&\& \ xs == ys \\ (a, c) \texttt{ == } (b, d) & = & a == b \ \&\& \ c == d \end{array}$$

$\Gamma_s = \{(\texttt{==}) : \forall\alpha.\{\alpha \to \alpha \to \mathit{Bool}\}.[\alpha] \to [\alpha] \to \mathit{Bool},\ \ (\texttt{==}) : \mathit{Int} \to \mathit{Int} \to \mathit{Bool},$
$\mathit{show} : \forall\alpha.\{\mathit{show} : \alpha \to \mathit{String}\}.[\alpha] \to \mathit{String}, \mathit{show} : \mathit{Int} \to \mathit{String}\}$

$\kappa = \{(\texttt{==}) : \alpha \to \alpha \to \mathit{Bool}, \mathit{show} : \beta \to \mathit{String}\}$

$\mathit{sats}(\kappa, \Gamma_s, \varkappa, \emptyset)$

- $\mathit{cSat} = \{(\texttt{==}) : \mathit{Int} \to \mathit{Int} \to \mathit{Bool}, \mathit{show} : \mathit{Int} \to \mathit{String}\}$
- $\mathit{sats}(\emptyset, \Gamma_s, \varkappa, \{(\texttt{==}) : \alpha \to \alpha \to \mathit{Bool}, \mathit{show} : \alpha \to \mathit{String}\})$

- $\mathit{cSat} = \{(\texttt{==}) : \{\alpha_1 \to \alpha_1 \to \mathit{Bool}\}.[\alpha_1] \to [\alpha_1] \to \mathit{Bool}, \mathit{show} : \mathit{Int} \to \mathit{Bool}\}$
- $\mathit{sats}(\{(\texttt{==}) : \alpha_1 \to \alpha_1 \to \mathit{Bool}\}, \Gamma_s^1, \varkappa, \kappa^1)$
  - $\mathit{cSat} = \{(\texttt{==}) : \mathit{Int} \to \mathit{Int} \to \mathit{Bool}\}$
  - $\mathit{sats}(\emptyset, \Gamma_s^1, \varkappa, \kappa^1)$

- $\mathit{cSat} = \{(\texttt{==}) :: \mathit{Int} \to \mathit{Int} \to \mathit{Bool}, \mathit{show} : \{\mathit{show} : \beta_1 \to \mathit{String}\}.[\beta_1] \to \mathit{String}\}$
- $\mathit{sats}(\{\mathit{show} : \beta_1 \to \mathit{String}\}, \Gamma_s^2, \varkappa, \kappa^2)$
  - $\mathit{cSat} = \{\mathit{show} : \mathit{Int} \to \mathit{String}\}\}$
  - $\mathit{sats}(\emptyset, \Gamma_s^2, \varkappa, \kappa'')$

- $\mathit{cSat} = \{(\texttt{==}) : \{\alpha_2 \to \alpha_2 \to \mathit{Bool}\}.[\alpha_2] \to [\alpha_2] \to \mathit{Bool},$
  $\mathit{show} : \{\mathit{show} : \beta_2 \to \mathit{String}\}.[\beta_2] \to \mathit{String}\}$
- $\mathit{sats}(\{(\texttt{==}) : \alpha_2 \to \alpha_2 \to \mathit{Bool}, \mathit{show} : \{\mathit{show} : \beta_2 \to \mathit{String}\}.[\beta_2] \to \mathit{String},$
  $\Gamma_s^3, \varkappa, \kappa^3)$
  - $\mathit{cSat} = \{(\texttt{==}) : \mathit{Int} \to \mathit{Int} \to \mathit{Bool}, \mathit{show} : \mathit{Int} \to \mathit{String}\}$
  - $\mathit{sats}(\emptyset, \Gamma_s^3, \varkappa, \kappa^3)$

Onde: $\Gamma_s^1 = \Gamma_s \ominus \{(\texttt{==}) : \{\alpha_1 \to \alpha_1 \to \mathit{Bool}\}.[\alpha_1] \to [\alpha_1] \to \mathit{Bool}\}$
$\Gamma_s^2 = \Gamma_s \ominus \{\mathit{show} : \{\mathit{show} : \beta_1 \to \mathit{String}\}.[\beta_1] \to \mathit{String}\}$
$\Gamma_s^3 = \Gamma_s \ominus \{(\texttt{==}) : \{\alpha_2 \to \alpha_2 \to \mathit{Bool}\}.[\alpha_2] \to [\alpha_2] \to \mathit{Bool},$
$\mathit{show} : \{\mathit{show} : \beta_2 \to \mathit{String}\}.[\beta_2] \to \mathit{String}\}$
$\kappa^1 = \kappa \oplus \{(\texttt{==}) : \{\alpha_1 \to \alpha_1 \to \mathit{Bool}\}$
$\kappa^2 = \kappa \oplus \{\mathit{show} : \beta_1 \to \mathit{String}\}$
$\kappa^3 = \kappa \oplus \{(\texttt{==}) : \alpha_2 \to \alpha_2 \to \mathit{Bool}, \mathit{show} : \beta_2 \to \mathit{String}\}$

Figura 4.3: Exemplo: Satisfazibilidade de 2 Restrições Independentes

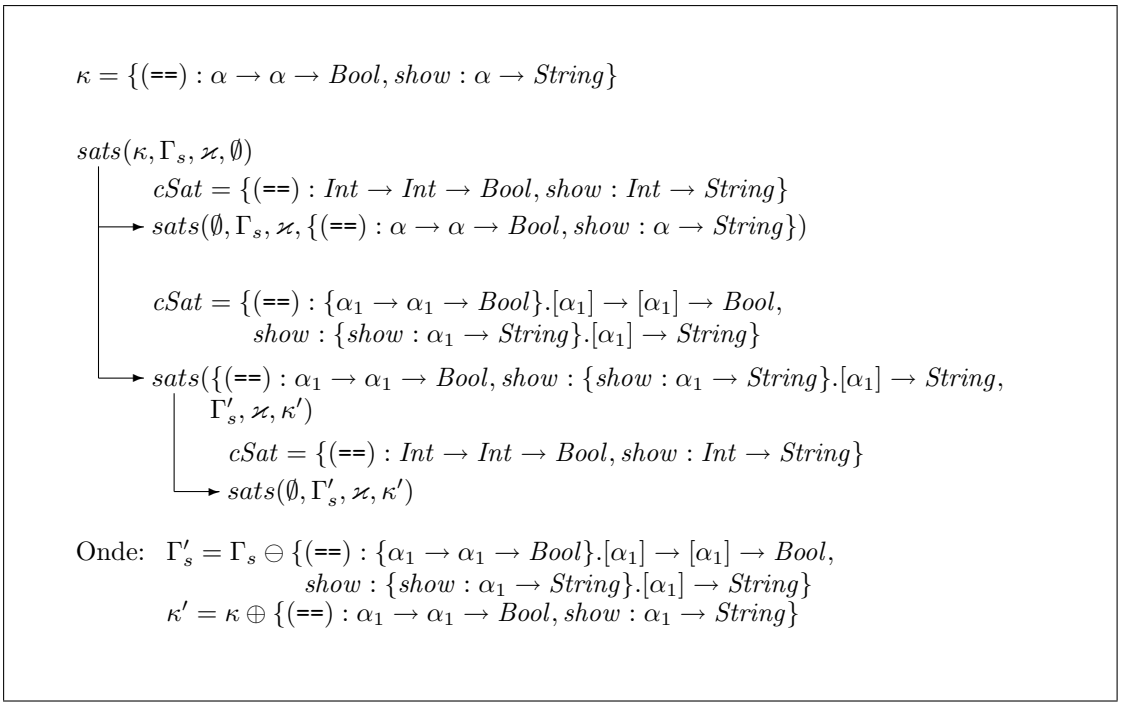

Figura 4.4: Exemplo: Satisfazibilidade de 2 Restrições com Dependência

O tipo inferido para cada uma dessas definições é:

$$(==) : Int \to Int \to Bool$$
$$(==) : \{(==) : a \to b \to Bool\}.[a] \to [b] \to Bool$$
$$(==) : \{(==) : a \to b \to Bool, (==) : c \to d \to Bool\}.(a, c) \to (b, d) \to Bool$$

A verificação da satisfazibilidade das restrições que ocorrem no tipo da definição de (==) para pares de valores envolve a verificação de duas restrições independentes, relativas ao próprio símbolo (==). O conjunto de suposições de tipo que satisfazem essas restrição, retornado na primeira chamada à função *sats*, tem tamanho n^2, onde n é o número de definições sobrecargas para o símbolo (==) (nesse caso, 3). Uma definição de (==) para uma m-tupla envolveria a verificação de m restrições independentes, relativas ao próprio símbolo (==), resultando em um conjunto *cSat* de tamanho n^m. Considerando que o processo de verificação de satisfazibilidade envolve a verificação das restrições de cada uma das suposições em *cSat*, é fácil ver que, mesmo com um valor de m não muito grande, a verificação de satisfazibilidade torna-se inviável na prática.

Esse problema pode ser, entretanto, facilmente contornado, observando-se que a verificação de satisfazibilidade de restrições que não possuem variáveis de tipo em comum não precisa ser tratada em conjunto. Portanto, é possível otimizar a implementação da função *sat*, separando as restrições em conjuntos independentes, em um processo que chamamos de projeção de restrições, que é

$$
\begin{array}{l}
\hat{\pi}(\emptyset) \qquad\qquad = \emptyset \\
\hat{\pi}(\{o:\tau\} \cup \kappa) \; = \texttt{let } \mathbb{K} = \hat{\pi}(\kappa) \texttt{ in} \\
\quad \texttt{if } \exists \kappa' \in \mathbb{K} \texttt{ tal que } tv(\tau) \cap tv(\kappa') \neq \emptyset \\
\qquad \texttt{then } \mathbb{K} - \{\kappa'\} \cup \{\kappa' \cup \{o:\tau\}\} \\
\qquad \texttt{else } \mathbb{K} \cup \{\{o:\tau\}\}
\end{array}
$$

Figura 4.5: Projeção de Restrições

$$
\begin{array}{l}
sat(\kappa, \Gamma) \qquad\qquad = sat_o(\kappa, \Gamma, \varkappa, \emptyset) \\
\\
sat_o(\kappa, \Gamma, \varkappa, \bar{\bar{\kappa_0}}) \;\; = S_1 \circ S_2 \circ \cdots S_n \\
\quad \text{onde} \;\; \{\kappa_i\}^{i=1..n} = \hat{\pi}(\kappa) \\
\qquad\qquad S_i = sats(\kappa_i, \Gamma, \varkappa, \bar{\bar{\kappa_0}})
\end{array}
$$

Figura 4.6: Satisfazibilidade de Projeções

descrito na Figura 4.5. Desse modo, para um conjunto de restrições $\{o_i : \tau_i\}$ que não possuem variáveis em comum, o tamanho de *cSat* seria dado por $t_1 + ... + t_n$, onde t_i é o número de suposições que satisfazem a restrição $o_i : \tau_i$. A função *sat* "otimizada"é dada na Figura 4.6.

Considerando ainda o exemplo anterior, pode parecer um pouco estranho, a principio, que o tipo inferido para a definição de (==) para lista não seja $\{(==) : \alpha \to \alpha \to Bool\}.[\alpha] \to [\alpha] \to Bool$. Isso ocorre porque, no momento da inferência do tipo dessa definição, não estão ainda disponíveis os tipos das demais definições sobrecarregadas para esse símbolo. Por esse motivo, o tipo requerido para o símbolo (==) na expressão x == y é inferido como sendo $\alpha \to \beta \to Bool$. O tipo correto é para essa definição de (==) é determinado apenas na fase de aperfeiçoamento de tipos, executada após a verificação de satisfazibilidade. De forma similar, o tipo da definição de (==) para pares é aperfeiçoado para: $\{(==) : \alpha \to \alpha \to Bool, (==) : \beta \to \beta \to Bool\}.(\alpha, \beta) \to (\alpha, \beta) \to Bool$.

4.3 Sintaxe

Na implementação do protótipo, procuramos introduzir o menor número possível de modificações na sintaxe original da linguagem *Haskell*, sendo entretanto necessárias as seguintes alterações:

– Declarações de classes e instâncias foram removidas, uma vez que a proposta do sistema *CT* é justamente eliminar a necessidade dessas declarações.
– Foi acrescentada a palavra reservada `overload`, que é usada para separar equações que consistem em definições sobrecarregadas consecutivas de um mesmo símbolo. Nos casos em que essa construção sintática não é usada, considera-se que essas equações consecutivas constituem partes de uma mesma definição para o símbolo.
– É prevista também a introdução da palavra reservada `assume`, para o especificação de tipos de símbolos sobrecarregados em uma abordagem de mundo aberto, conforme discutido na Seção 3.5.
– Nas anotações de tipos, a declaração explícita de restrições foi removida, sendo essas restrições automaticamente determinadas pelo processo de inferência de tipos.

O uso de uma anotação de tipos possibilita ao programador restringir o tipo de uma expressão para um tipo que é uma instância do tipo principal dessa expressão. Isso pode ser necessário para resolver problemas de ambigüidade e para resolução de sobrecarga no caso de existência de instâncias sobrepostas.

Nessa implementação, anotações de tipos também podem ser usadas para separar duas definições sobrecarregadas consecutivas, eliminando, neste caso, a necessidade de uso da palavra reservada `overload`. No exemplo a seguir, as declarações seriam rejeitas, caso não fosse introduzida a anotação de tipos, por apresentar sobreposição dos tipos:

$$double\ (x : xs)\ \texttt{=}\ (double\ \ x : double\ \ xs)$$

$$double :: Int \rightarrow Int$$
$$double\ \ x\ \texttt{=}\ x * x$$

Se quisermos que a função *double* seja também sobrecarregada para o tipo *Float*, será necessário fazer uma nova declaração (que pode ser idêntica a declaração para o tipo *Int*), com a anotação de tipo *double* ::*Float*→*Float*. A linguagem *Haskell* também apresenta essa limitação, obrigando o programador a fazer duas declarações, nesse caso de instâncias da classe que define a sobrecarga dessa função, uma para o tipo *Int* e outra para o tipo *Float*. Declarações como do exemplo abaixo não são permitidas em *Haskell*:

```
class D α where
   double :: α → α

instance (Num α) ⇒ D α where
   double x = x ∗ x     {- Declaração Inválida -}
```

4.4 Resultados

Os principais objetivos a serem atingidos com a implementação do protótipo foram inicialmente definidos como:

- Analisar a viabilidade e o comportamento do algoritmo de verificação da satisfazibilidade, utilizando como entrada código já implementado em *Haskell*.
- Analisar o comportamento dos algoritmos de inferência e de verificação de satisfazibilidade na presença de declarações que envolvam sobrecarga e recursão polimórfica.
- Determinar possíveis necessidades e oportunidades de otimizações a serem introduzidas em uma implementação mais robusta, pela análise dos tempos de execução envolvidos em cada uma das fases de inferência de tipos.
- Comparar a eficiência do sistema *CT* com o sistema de classes de tipos de *Haskell*.

Os primeiros resultados obtidos sugerem que uma versão da linguagem *Haskell* sem classes de tipos é viável, e que, com a introdução de algumas otimizações, podemos atingir resultados bastante satisfatórios para códigos similares aos implementados hoje em *Haskell*. Além dos exemplos apresentados nesse trabalho, vários fragmentos de código e versões da biblioteca padrão de *Haskell* foram usados na fase de testes.

Os resultados apresentados a seguir são referentes à inferência de tipos de declarações similares às encontradas nos módulos *Char*, *Enum*, *Ix*, *Maybe*, *List* e na parte da biblioteca matemática que envolve números reais. Os dados foram obtidos por meio da ferramenta de verificação do perfil de execução de um programa disponível no código gerado pelo compilador *GHC* [1], executado em computador com processador *Pentium* de $1.1Ghz$ com $512Mb$ de memória.

Na Tabela 4.1, são apresentados os tempos de execução e os tamanhos da área de memória utilizada, para 3 diferentes execuções do programa, tendo como entrada 3 diferentes combinações dos módulos citados acima. Essa tabela

	Módulos	Tempo (*seg.*)	Memória (*Mb*)	Nº Decl.
Teste 1	*Char, Maybe, List, Enum, Ix* e *Math*	25	775	265
Teste 2	*Char, Maybe, Enum* e *List*	16	500	183
Teste 3	*Char, Maybe, Enum* e *Math*	5	175	144

	Teste 1		**Teste 2**		**Teste 3**	
Procedimento	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.
Análise Sintática	4,4%	7,0%	6,1%	8,6%	10,4%	15,4%
Inf. da Tipagem Principal	3,5%	3,9%	5,1%	4,7%	6,2%	8,1%
Semi-Unificação (Inf. - Req.)	72,1%	69,2%	66,2%	63,7%	46,9%	43,2%
Satisfazibilidade	9,6%	10,2%	12,9%	14,3%	17,8%	16,4%
Teste de Ambigüidade	1,4%	1,4%	1,7%	2,2%	2,5%	2,2%
Outros Procedimentos	9,0%	8,3%	8,0%	6,5%	16,2%	14,7%

Tabela 4.1: Perfil de Execução do *Front-end*

	Teste 1		**Teste 2**		**Teste 3**	
Procedimento	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.
Análise Sintática	4,3%	7,0%	6,6%	8,6%	11,4%	15,4%
Inf. da Tipagem Principal	3,2%	3,6%	3,8%	4,4%	6,1%	8,1%
Semi-Unificação (Inf. - Req.)	42,4%	36,1%	40,0%	34,5%	24,4%	24,4%
Satisfazibilidade	8,1%	9,8%	11,9%	13,9%	14,2%	15,7%
Teste de Ambigüidade	1,4%	1,4%	1,7%	2,2%	2,4%	2,2%
Outros Procedimentos	9,5%	8,3%	5,6%	6,4%	15,7%	14,2%
Unificação	31,1%	33,8%	30,4%	30,0%	25,8%	20,0%

Tabela 4.2: Perfil de Execução do *Front-end* destacando os recursos gastos em Unificações

descreve os recursos usados pelos principais processos envolvidos na inferência de tipos. Pode-se se observar que o aumento do número de declarações tem impacto maior no processo de semi-unificação de tipos inferidos e requeridos, sendo esse o processo que mostrou demandar maior tempo em quase todos os testes realizados.

Analisando os dados apresentados, verifica-se também que grande parte do tempo de execução é consumido em unificações de tipos. Para a maior parte dos códigos usados para teste do processo de inferência de tipos, o tempo consumido em unificações constitui aproximadamente 30% do tempo total de execução. O consumo percentual de memória nesse processo é também em torno de 30% do consumo de memória total. Isso sugere que uma primeira melhoria a ser introduzida é a implementação de um algoritmo de unificação de tempo linear [47], em substituição ao código implementado para unificação, o qual baseia-se em um algoritmo de complexidade exponencial, uma vez que, nesse primeiro momento da implementação do protótipo a preocupação maior era com a simplicidade e não com eficiência.

Na Tabela 4.2, o percentual de tempo e a memória utilizada em unificações é computado separadamente. Pode-se observar que as unificações têm maior impacto justamente no processo de semi-unificação, não sendo significativo no processo de verificação de satisfazibilidade, e praticamente desprezível nos demais processos.

Apesar de verificarmos que a implementação inicial do processo de semi-unificação ficou bastante ineficiente, não foram detectados maiores problemas na inferência de tipos de declarações que envolvam recursão polimórfica em conjunto com sobrecarga. Um exemplo no qual esses dois recursos são utilizados é apresentado a seguir:

$$\texttt{data}\ Seq\ t\ \texttt{=}\ Nil\ \texttt{|}\ Cons\ t\ (Seq(t,t))$$

$$\begin{aligned} double\ Nil\ &\texttt{=}\ Nil \\ double\ (Cons\ x\ s)\ &\texttt{=}\ Cons\ (double\ x)\ (double\ s) \end{aligned}$$

$$\begin{aligned} &double :: Int \rightarrow Int \\ &double\ x\ \texttt{=}\ x * x \end{aligned}$$

A definição sobrecarregada da função *double* que envolve recursão polimórfica tem seu tipo inferido corretamente como:

$$double : \{double : \alpha \rightarrow \alpha\}.Seq\ \alpha \rightarrow Seq\ \alpha$$

O comportamento da inferência de tipos para definições como acima estimulam um maior investimento na otimização da implementação. Como o tempo gasto pelo protótipo ainda é bastante superior aos tempos apresentados por interpretadores e compiladores disponíveis para a linguagem *Haskell*, tendo como entrada um código similar, não incluímos aqui dados comparativos entre os dois sistemas. Por se tratar de um primeiro protótipo, onde a questão de eficiência não foi prioritária, esse desempenho não chega a ser surpresa, sendo sua otimização planejada como um dos trabalhos futuros.

5 Conclusão

O sistema CT estende o sistema Damas-Milner com o suporte a sobrecarga de símbolos, mantendo características centrais desse sistema, como a inferência de tipos principais de expressões baseada no conjunto de nomes visíveis no contexto onde essas expressões ocorrem. A política de tratamento de sobrecarga adotada no sistema CT é muito menos restritiva do que as encontradas em sistemas similares. O sistema CT foi originalmente proposto em [7], onde o problema da satisfazibilidade de restrições foi incorretamente apresentado como sendo decidível. Nesse trabalho é apresentada uma revisão do sistema CT, no qual o problema da satisfazibilidade de restrições é definido de forma independente do sistema de tipos. É apresentado um algoritmo para solução desse problema, que usa um limite de iterações para tratar os casos em que o problema não pode ser decidido.

É apresentado também um protótipo de implementação desse algoritmo. Os primeiros resultados obtidos com o uso desse protótipo, por meio de testes feitos com códigos similares a códigos implementados em *Haskell*, corroboram a idéia de que o sistema CT pode funcionar bem, como base para definição de uma linguagem de programação real. Para estes casos, o limite de iterações não foi necessário em nenhuma situação. Entretanto, uma idéia mais realista só poderá ser de fato obtida quando tivermos terminado a implementação de um interpretador ou compilador definido com base nesse sistema, uma vez que é natural esperar que, em conseqüência de um aumento na facilidade da declaração de símbolos sobrecarregados, o uso desse tipo de polimorfismo venha a ser significativamente aumentado, em comparação com os programas implementados atualmente em *Haskell*.

A continuação desse trabalho envolve tópicos como os seguintes:

- implementação de um *back-end* ou alteração do *front-end* do protótipo implementado para que possa ser integrado a um *back-end* de um compilador *Haskell* já existente, como por exemplo o *GHC*;
- aprimoramento da implementação do protótipo desenvolvido, tornando-o mais eficiente e com um melhor tratamento de erros;

– alteração dos algoritmos de inferência de tipos e de teste da satisfazibilidade de restrições de tipos para suporte a abordagem de mundo aberto;
– extensão do sistema *CT* para o tratamento de sobrecarga de campos de registro;
– estudo de formas de incluir suporte a programação genérica, tipicamente envolvendo uma cláusula similar à cláusula `derive`, existente em *Haskell*, mas de maneira mais abrangente.

Bibliografia

[1] The Glasgow Haskell Compiler User's Guide, Version 6.2. Technical report, The University Court of the University of Glasgow, 2002.

[2] J. Tiuryn A. J. Kfoury and P. Urzyczyn. The Undecidability of the Semi-Unification Problem. *Information and Computation*, 102(1):83–101, January 1993.

[3] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, second edition, 1996.

[4] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, third edition, 2000.

[5] R. Bird and L. Meertens. Nested Datatypes. In *Proceedings of the 4th International Conference on Mathematics of Program Construction, Springer-Verlag LNCS 1422*, 1998.

[6] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Europe, second edition, 1998.

[7] Carlos Camarão and Lucília Figueiredo. Type Inference for Overloading without Restrictions, Declarations or Annotations. In *Proc. of FLOPS´99, LNCS 1722*, pages 37–52, 1999.

[8] Carlos Camarão, Lucília Figueiredo, and Elaine Pimentel. Sistemas de Tipos em Linguagens de Programação. In *III Brazilian Symposium on Programming Languages (SBLP'99)*, 1999.

[9] Carlos Camarão, Lucília Figueiredo, and Cristiano D. Vasconcellos. Constraint-Set Satisfiability for Overloading. In *6th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*. ACM Press, 2004.

[10] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17:471–522, 1985.

[11] Kung Chen, Paul Hudak, and Martin Odersky. Parametric Type Classes. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 170–181. ACM Press, 1992.

[12] R. H. Connelly and F. L. Morris. A Generalisation of the Trie Data Structure. 6(1):1–28, 1995.

[13] Luis Damas and Robin Milner. Principal Type-Schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–212. ACM Press, 1982.

[14] Dominic Duggan and John Opel. Type-Checking Multi-Parameter Type Classes. *Journal of Functional Programming*, 12(2):133–158, 2002.

[15] F. Henderson et al. The mercury project. http://www.cs.mu.oz.au/research/mercury/.

[16] Simom Peyton Jones et al. *Haskell 98 Language and Libraries The Revised Report*. Cambridge University Press, 2003.

[17] Karl-Filip Faxen. Haskell and Principal Types. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 88–97. ACM Press, 2003.

[18] Lucília Figueiredo and Carlos Camarão. Principal Typing and Mutual Recursion. In *International Workshop on Functional and Logic Programming (WFLP'2001)*, 2001.

[19] Lucília Figueiredo and Carlos Camarão. A View on Abstract and Extensible Types. *Revista Colombiana de Computación*, 3(1):21–40, 2002.

[20] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type Classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.

[21] Fritz Henglein. Type Inference with Polymorphic Recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.

[22] Said Jahama and A. J. Kfoury. A general theory of semi-unification. Technical Report BU-CS 1993-018, 1993.

[23] Kathleen Jensen and Niklaus Wirth. *PASCAL - User Manual and Report*. Springer Verlag, 1974.

[24] Trevor Jim. What are Principal Typings and what are they good for? In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 42–53. ACM Press, 1996.

[25] Mark Jones. *Qualified Types: Theory and Pratice*. Cambridge Press, 1994.

[26] Mark P. Jones. Coherence for Qualified Types. Technical Report YALEU/DCS/RR-989, Yale University, 1993.

[27] Mark P. Jones. Simplifying and Improving Qualified Types. In *Proceedings of the seventh International Conference on Functional Programming Languages and Computer Architecture*, pages 160–169. ACM Press, 1995.

[28] Mark P. Jones. Typing Haskell in Haskell. Technical Report UU-CS-1999-28, University of Utrecht, 1999.

[29] Mark P. Jones. Type Classes with Functional Dependencies. In *In Proceedings of the 9th European Symposium on Programming*, 2000.

[30] Simon Peyton Jones and Mark P. Jones. Type Classes: an exploration of the desing space. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, 1997.

[31] Stefan Kaes. Parametric Overloading in Polymorphic Programming Languages. In *Proceedings of the 2nd European Symposium on Programming*, pages 131–144. Springer-Verlag, 1988.

[32] P. C. Kanellakis and J. C. Mitchell. Polymorphic Unification and ML Typing. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 105–115. ACM Press, 1989.

[33] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[34] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type Teconstruction in the Presence of Polymorphic Recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, 1993.

[35] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. An Analysis of ML Typability. *J. ACM*, 41(2):368–398, 1994.

[36] Daan Leijen and Erik Meijer. Parsec: Direct Style Monadic Parser Combinators for the Real World. Technical Report UU-CS-2001-35, Departement of Computer Science, Universiteit Utrecht, 2001.

[37] Alberto Martelli and Ugo Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[38] John McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. *Lisp 1.5 Programmer's Manual*. MIT Press, second edition, 1965.

[39] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and Systems Sciences*, (17):384–375, 1978.

[40] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[41] John Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[42] Alan Mycroft. Polymorphic Type Schemes and Recursive Definitions. In *Proceedings of the 6th Colloquium on International Symposium on Programming*, pages 217–228. Springer-Verlag, 1984.

[43] Tobias Nipkow and Christian Prehofer. Type Reconstruction for Type Classes. *Journal of Functional Programming*, 5(2):201–224, 1995.

[44] Martin Odersky, Philip Wadler, and Martin Wehr. A Second look at Overloading. In *Proceedings of the seventh International Conference on Functional Programming Languages and Computer Architecture*, pages 135–146. ACM Press, 1995.

[45] Chris Okasaki. Catenable double-ended queues. In *Proceedings of the second ACM SIGPLAN International Conference on Functional Programming*, pages 66–74. ACM Press, 1997.

[46] Chris Okasaki. *Purely Functional Data Structures*. Cambridge Press, 1998.

[47] M. S. Paterson and M. N. Wegman. Linear Unification. In *Proceedings of the eighth annual ACM Symposium on Theory of Computing*, pages 181–186. ACM Press, 1976.

[48] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[49] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965.

[50] Mark Shields and Simon Peyton Jones. Object-Oriented Style Overloading for Haskell. In Nick Benton and Andrew Kennedy, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier, 2001.

[51] Geoffrey S. Smith. *Polymorphic Type Inference for Languages with Overloading and Subtyping*. PhD thesis, Department of Computer Science, Cornell University, 1991.

[52] Geoffrey S. Smith. Principal Type Schemes for Functional Programs with Overloading and Subtyping. In *Selected papers of the Colloquium on Formal Approaches of Software Engineering*, pages 197–226. Elsevier Science Publishers B. V., 1994.

[53] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 2000.

[54] Peter J. Stuckey and Martin Sulzmann. A Theory of Overloading. In *Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*, pages 167–178. ACM Press, 2002.

[55] Cristiano Vasconcellos, Lucília Figueiredo, and Carlos Camarão. Practical Type Inference for Polymorphic Recursion: an Implementation in Haskell. *j-jucs*, 9(8):873–890, aug 2003. `http://www.jucs.org/jucs_9_8/practical_type_inference_for`.

[56] Dennis M. Volpano. Haskell-style Overloading is NP-hard. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 88–94, 1994.

[57] Dennis M. Volpano and Geoffrey Smith. On the Complexity of ML Typability with Overloading. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 15–28. Springer-Verlag, 1991.

[58] P. Wadler and S. Blott. How to make ad-hoc Polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 60–76. ACM Press, 1989.