António Mário da Silva Marcos Florido

Sistemas de Tipos para Linguagens Declarativas

Tese submetida para obtenção
do grau de Doutor em Ciência de Computadores
Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
1997

À Teresa e aos meus Pais.

Agradecimentos

Em primeiro lugar quero agradecer ao meu orientador, Prof. Luís Damas, pela apresentação do tema, ideias e discussões, que contribuiram de um modo determinante para a elaboração deste trabalho.

Um agradecimento tambem para muitos colegas do Laboratório de Inteligência Artificial e de Ciência de Computadores da UP por terem proporcionado um excelente ambiente de trabalho durante todos estes anos. Desculpando desde já todos os que não refiro, há alguns nomes que me sinto obrigado a enumerar: A Sabine pela paciência e dedicação ao lêr a minha tese, contribuindo com muitas indicações úteis, ao Zé Paulo pelas inúmeras conversas sobre todos os temas (incluindo trabalho!) durante todos estes anos, ao Alípio pela amizade constante e ao Victor Costa, Armando Matos, Ana Paula, Miguel Filgueiras, Fernando Silva, Nelma, Rogério e Eduardo (têr um amigo de longa data como colega de gabinete é sem dúvida um privilégio) os meus mais sinceros agradecimentos.

Um abraço também para o Paulo Azevedo e para o José João Almeida. Esta tese deve muito a muitas conversas mantidas com ambos.

O meu trabalho foi enriquecido por varias opiniões. Não quero deixar de agradecer as de Mario Coppo e Stefan Van Bakel sobre sistemas de tipos com intersecções.

Agradeço também à JNICT pelo apoio financeiro concretizado em duas bolsas de estudo, do programa Ciência (BD/1330/91 - IA) e Praxis XXI (BD/3287/94), que foram fundamentais para durante todos estes anos ter podido dedicar-me exclusivamente ao trabalho de investigação de que resultou esta tese de doutoramento.

Um agradecimento muito especial também para os meus pais. Força de vontade, humildade e sentido de auto-crítica foram valores que eles me transmitiram ao longo dos anos, valores esses que se revelaram fundamentais quando se trabalha em investigação. Um agradecimento muito especial à Teresa pela paciência e apoio constante durante todos estes anos.

Índice

Resumo

Esta tese tem como objectivo desenvolver caracterizações semânticas precisas de sistemas de tipos descritivos para linguagens declarativas.

A nossa tese é a de que as relações sintácticas entre tipos e termos resultam de relações semânticas muito mais fortes do que a integridade, usualmente exigida aos sistemas de tipos. Temos como interesse principal definir os sistemas de tipos a partir de relações semânticas entre tipos e termos e não definir os sistemas e, posteriormente, arranjar uma semântica que garanta boas propriedades.

O critério semântico usado normalmente para relacionar termos e tipos é a integridade. Essa relação semântica é apresentada para os seguintes sistemas de tipos:

1. O sistema de tipos simples para o λ-calculus;
2. O sistema de tipos da linguagem ML;
3. Os sistemas de tipos com intersecções;
4. Várias metodologias para a definição de sistemas de tipos no paradigma da programação em lógica.

Baseados na relação semântica entre termos e tipos definimos, para cada termo, um conjunto de restrições nos tipos que garantem a sua validade semântica.

Atendendo à visão de Curry de tipos como descrições, definimos condições semânticas necessárias ao carácter descritivo dos tipos. Demonstramos que os sistemas de Curry e da linguagem ML são integros com respeito a essa caracterização semântica. O mesmo não acontece nos sistemas de tipos com intersecções, e usando essa propriedade explicamos, sob o ponto de vista semântico, algumas anomalias destes sistemas.

Baseados em propriedades semânticas dos sistemas de tipos identificadas no λ-calculus, definimos um critério semântico de classificação de tipos para programas em lógica. Usamos esse critério para demonstrar a adequação de uma linguagem: a linguagem regular determinística, e a não adequação de outra: a linguagem regular, para denotar tipos de programas em lógica.

Capítulo 1

Introdução

O objectivo desta tese é desenvolver a base de uma caracterização semântica de sistemas de tipos descritivos para linguagens declarativas.

A única propriedade semântica imposta aos sistemas de tipos é a integridade. Nós pensamos que esta propriedade por si só não basta para garantir que os tipos derivados para um determinado termo sejam descrições úteis desse termo. Para conseguir o objectivo proposto, estudámos e desenvolvemos novas metodologias aplicadas a dois casos diferentes:

1. O λ-calculus, dado tratar-se do laboratório por excelência para o estudo e desenvolvimento de sistemas de tipos;

2. A programação em lógica, porque se trata de um paradigma declarativo onde os tipos assumem uma noção particular e onde os sistemas desenvolvidos até agora estão longe de conduzir a uma metodologia geral.

No λ-calculus focámos a nossa atenção em três sistemas: o sistema de Curry, o sistema polimórfico de Damas-Milner e o sistema com intersecções de Coppo-Dezanni. A nossa escolha foi motivada pelo facto de cada um deles ser uma extensão do anterior com objectivos bem precisos e, portanto, passíveis de um estudo sistemático. Além disso, os dois primeiros estão na base de sistemas usados na prática em linguagens de programação funcional e o terceiro promete resolver alguns problemas inerentes a estas linguagens. Nós tivemos, desde sempre, como motivação a aplicabilidade futura dos resultados teóricos obtidos, na área de implementação de linguagens de programação.

Na programação em lógica, desenvolvemos uma metodologia própria com o objectivo de unificar os sistemas existentes e usámos essa metodologia para o estudo semântico efectuado posteriormente.

Foi nosso objectivo tratar o tema proposto em paradigmas diferentes de programação declarativa e, dentro de cada um deles, em diversos sistemas de tipos existentes.

1.1 Tipos em Linguagens de Programação

No desenho e concepção de uma linguagem de programação, podemos optar pela declaração de tipos como parte integrante da linguagem, dando origem a linguagens tipificadas como o *C* ou o *Pascal*, ou pela omissão de tais declarações, dando origem a linguagens onde não declaramos tipos, como o *Prolog*.

Em defesa das linguagens tipificadas surgem argumentos como:

1. A especificação mais natural dos problemas tem em conta os tipos dos objectos;
2. É mais fácil proceder à verificação da correcção dos algoritmos;
3. Permite detectar muitos erros de incompatibilidade de tipos em tempo de compilação;
4. Permite tirar partido de técnicas de optimização em tempo de compilação.

O uso de linguagens não tipificadas é defendido tendo em conta argumentos como:

1. Maior liberdade e, portanto, expressividade das linguagens;
2. Maior facilidade de escrita dos programas, sem ser obrigatória a escrita de declarações de tipos que, por vezes, são óbvias;
3. O facto de, em muitos casos, ser possível inferir os tipos de programas não tipificados em tempo de compilação, mantendo assim a capacidade de detectar inconsistências de tipos e de efectuar optimizações no processo de compilação.

Sob o ponto de vista semântico, as linguagens tipificadas têm uma semântica tipificada. Nas linguagens não tipificadas, os tipos têm uma semântica própria, independente da semântica do programa, e é estabelecida uma relação semântica entre tipos e objectos que está ligada ao conceito de tipificação. Neste último caso, os tipos são vistos como propriedades de objectos não tipificados, contrastando com as semânticas tipificadas onde os objectos só têm significado quando vistos como elementos de um determinado tipo. Como exemplo, considere-se a função identidade $\lambda x.x$ com tipo $int \to int$. Numa semântica tipificada a identidade é restringida aos números inteiros. Numa semântica não tipificada a identidade é uma função que, quando aplicada a qualquer objecto, retorna o mesmo objecto, e que possui a propriedade $int \to int$, isto é, quando a aplicamos a um número inteiro, retorna um número inteiro.

Em 1932, Alonzo Church desenvolveu um modelo das funções computáveis, o λ-calculus, que mais tarde veio dar origem ao paradigma da programação funcional. Mais tarde, [Chu40], apresentou uma versão do λ-calculus em que as variáveis eram anotadas com os seus tipos. Ao denotar uma função, temos que, obrigatoriamente, explicitar o seu domínio. Isto é conseguido à custa de um conjunto de termos anotados em que as variáveis têm a forma x^τ, com o significado x tem tipo τ, e as funções têm a forma $\lambda x^\tau.M$. Neste sistema, se o tipo de M fôr β o tipo de $\lambda x^\tau.M$ é, obrigatoriamente, $\tau \to \beta$. Esta versão tipificada do λ-calculus modela as linguagens tipificadas.

Em [Cur34] Curry apresentou tipos de um modo diferente. Esta nova visão de tipos, quando aplicada ao λ-calculus consiste no seguinte: temos o λ-calculus constituido por um conjunto de termos. Temos depois um conjunto de regras que, a partir da estrutura sintáctica dos termos, permitem inferir o seu carácter funcional. Por exemplo, o facto do termo MN ter um tipo β é implicado pela existência de um tipo α para N tal que $\alpha \to \beta$ seja o tipo de M. Esta visão de tipos deu origem à inferência de tipos para linguagens cuja semântica não é tipificada. No decorrer deste trabalho, vamos assumir a visão de Curry, ou seja, para nós, os objectos pertencem a domínios semânticos não tipificados e os tipos são propriedades que pretendemos inferir a partir da estrutura sintáctica desses objectos.

Vários sistemas de tipos resultam de extensões ao sistema original de Curry e da aplicação deste sistema a paradigmas diferentes duma visão funcional. O número de sistemas de tipos resultantes é enorme, e verifica-se que muitos deles assentam em extensões ao sistema original de Curry, seguindo critérios ad-hoc para resolver este ou aquele problema específico, o que, em muitos casos, originou problemas, por vezes, bem mais graves.

1.2 Propriedades Fundamentais dos Sistemas de Tipos

O nosso objectivo é identificar, atendendo à visão original de Curry de que um tipo é uma descrição de um objecto, um certo número de propriedades que devem ser respeitadas por qualquer sistema de tipos descritivos.

Mostramos que algumas dessas propriedades não são respeitadas por algumas extensões bem simples do sistema de Curry, como por exemplo, os sistemas de tipos com intersecções. Aplicamos algumas dessas propriedades ao estudo de tipos num paradigma diferente do funcional, a programação em lógica.

Vamos agora apresentar e ver em mais detalhe cada uma dessas propriedades:

1. *Inclusão semântica:* Num sistema de tipos descritivos a primeira propriedade que temos que ter em conta é a de que os tipos sejam de facto descrições dos objectos. Essas descrições são, sob o ponto de vista semântico, aproximações dos objectos, no sentido em que um tipo pode descrever vários objectos. Sendo assim, a semântica de um tipo τ, $\|\tau\|$, deve ser constituída pelo conjunto das semânticas dos objectos que descreve. A propriedade da inclusão semântica é satisfeita definindo, como é habitual, um conceito semântico de tipificação que corresponde à inclusão semântica. Existe uma relação, $\models M : \tau$, entre a semântica de um objecto $\|M\|$ e a do tipo, $\|\tau\|$, que é definida como:

$$\models M : \tau \Leftrightarrow \|M\| \in \|\tau\|$$

   Esta propriedade é fundamental, existe e é aceite por todos os sistemas de tipos descritivos. No paradigma da programação em lógica, assume um papel preponderante dado ser a base dos sistemas de inferência de tipos; nesse paradigma, um tipo é uma aproximação conservativa da semântica do programa.

2. *Integridade forte:* A propriedade anterior é usada na definição da relação semântica entre tipos e objectos. No entanto, os primeiros sistemas definem outra relação entre tipos e objectos, $\vdash M : \tau$, através de um conjunto de regras de inferência com o seguinte significado: $\vdash M : \tau$ se é possível inferir τ para M usando o conjunto de regras que definem o sistema de tipos. Para garantir que os tipos inferidos para um objecto são de facto descrições desse objecto temos que exigir a integridade do sistema, ou seja:

$$\vdash M : \tau \Rightarrow \models M : \tau$$

Sob o ponto de vista semântico, a relação $\vdash$ está contida na relação $\models$. A condição anterior, exigida por todos os sistemas de tipos descritivos, é uma integridade fraca. Fraca, no sentido em que, se exigirmos somente essa condição, podemos estar a admitir tipos que são descrições gerais demais. Como exemplo, se considerarmos sistemas com um tipo universal ω, o facto, semânticamente íntegro, de dizermos que todo o termo tem tipo ω, não nos dá nenhuma descrição útil do termo.

Um dos objectivos deste trabalho foi o de tentar caracterizar, sob o ponto de vista semântico, a relação $\vdash$, isto é, porque é que os sistemas de tipos, de entre todos os tipos admissíveis sob o ponto de vista semântico para um dado termo escolhem um determinado subconjunto.

Facilmente se verifica que a definição da relação $\vdash$ está ligada à sintaxe do termo. Neste ponto, levantam-se as seguintes questões:

(a) Quais as relações semânticas entre tipos e termos que correspondem às relações sintácticas definidas pelos sistemas?

(b) Essa relações semânticas são de facto definidas implicitamente pelo sistema de tipos ou deve, ao contrário, ser o sistema de tipos a respeitar uma relação semântica entre tipos e termos, mais forte do que a integridade, e que de algum modo denote o objectivo do sistema?

Vamos começar por abordar a segunda questão. Ao definir um sistema de tipos, tomam-se opções motivadas por aquilo que a nível semântico se espera do sistema. Considere-se o exemplo seguinte: o sistema de Curry tipifica a função identidade $\lambda x.x$ com o tipo $int \rightarrow int$. Partindo do princípio que existe uma relação semântica entre os tipos int e $real$ tal que $int \sqsubseteq real$ temos $\models \lambda x.x : int \rightarrow real$. A identidade é de facto uma função que, quando aplicada a um número inteiro, retorna um número real. No entanto, o sistema de Curry rejeita o tipo $int \rightarrow real$ para a identidade.

A nossa tese é que isto acontece, porque o objectivo do sistema de Curry é descrever tipos monomórficos, isto é, cada tipo representa-se apenas a si próprio, e não a outros tipos que, eventualmente, também sejam semanticamente admissíveis para o termo. Voltando ao exemplo anterior, repare-se que $\vdash \lambda x.x : int \rightarrow real$ pressupõe que $\{x : int\} \vdash x : real$, o que, apesar de ser válido sob o ponto de vista semântico, foge ao pressuposto do monomorfismo, dado que neste caso, do tipo *int* para x poderíamos inferir outros tipos para o mesmo termo x. De um modo geral, podemos concluir que:

$$\forall_{\alpha,\beta}.\{x : \alpha\} \models x : \beta$$

desde que $\|\alpha\| \subseteq \|\beta\|$. Como vamos ver no decorrer da tese, essas restrições podem ser definidas para todos os termos.

Voltando à motivação inicial do sistema de Curry, que é a de fornecer o carácter funcional dos termos numa perspectiva monomórfica, esse sistema só deve permitir a escolha de uma solução de $\|\alpha\| \subseteq \|\beta\|$. Além disso, deve escolher a solução que forneça mais informação sobre o termo, neste caso $\beta = \alpha$. Como vimos, chegámos a uma regra do sistema de tipos a partir de propriedades semânticas impostas pela motivação inicial do sistema. Vamos ver, no decorrer da tese, que essa caracterização precisa de algumas propriedades semânticas que qualquer sistema deve respeitar é possível para vários sistemas.

Se conseguirmos obter uma caracterização semântica precisa do objectivo descritivo do sistema de tipos, é válido exigirmos que o sistema seja íntegro com respeito ao seu objectivo inicial. A essa integridade chamamos integridade forte.

3. *Consistência em relação à derivação:* O nosso objectivo é estudar sistemas de tipos para linguagens de programação. Subjacente a qualquer linguagem de programação, mesmo às linguagens declarativas, existe uma semântica operacional que define o modo como o programa é executado. Essa semântica operacional define um conjunto de operações que, quando aplicadas a um programa, definem exactamente a sua execução. Cada operação é vista como um passo de computação. É desejável, permitimo-nos dizer obrigatório, que, se um programa P tiver um tipo τ, após um passo de computação, o resultado seja também tipificado por τ. Esta propriedade garante a tipificagem estática do programa, isto é, qualquer descrição do programa P é válida após qualquer execução de P. No λ-calculus essa propriedade equivale a manter os tipos após uma redução β, isto é, $\vdash M : \tau \wedge M \rightarrow_\beta N \Rightarrow \vdash N : \tau$. Em programação em lógica, esta propriedade equivale à nossa definição de tipos consistentes, isto é, um tipo é consistente com um programa se, para cada golo G, tipificar todos os subgolos necessários à demonstração de G.

1.3 Polimorfismo

No contexto de tipos descritivos, cada objecto pode ter vários tipos (várias descrições). Um tipo que denote um conjunto de tipos de um determinado objecto é chamado um tipo polimórfico.

Há várias formas de polimorfismo. Por exemplo, nas linguagens de programação por objectos, a herança nos tipos denota uma forma de polimorfismo, o polimorfismo por inclusão. Ao permitir que operadores aritméticos sejam aplicados a inteiros e reais temos outra forma de polimorfismo.

Quando uma linguagem funcional tipifica a função identidade, queremos que o tipo seja da forma $\alpha \rightarrow \alpha$ para qualquer substituição da variável α, ou seja, o tipo polimórfico de $\lambda x.x$ é $\forall_\alpha.\alpha \rightarrow \alpha$. Quando dizemos que o primeiro argumento do predicado *append* é do tipo $lista$, referimo-nos implicitamente a listas de qualquer elemento, isto é:

$$\begin{array}{l} list([]). \\ list([X|R]) \leftarrow \alpha(X), list(R). \end{array}$$

para qualquer substituição da variável α. O polimorfismo apresentado nestes dois últimos exemplos é conseguido à custa de parâmetros (variáveis nos tipos) e de operações, neste caso, a substituição, que permitem obter os diversos tipos admissíveis para um objecto a partir do tipo polimórfico. Por isso, é conhecido como polimorfismo paramétrico e é a única forma de polimorfismo que vamos tratar neste trabalho. Neste contexto, põe-se o seguinte problema: dado um objecto, existe algum tipo polimórfico a partir do qual podemos derivar qualquer tipo admissível para esse objecto? Esta propriedade é a propriedade do tipo principal e é extremamente importante no contexto de sistemas de tipos descritivos, pelas seguintes razões:

1. Garante uma estrutura uniforme entre todos os tipos admissíveis por um dado objecto, estrutura essa unificável no tipo principal;

2. Dado um tipo e um objecto, a verificação de tipos consiste em saber se o tipo é derivável do tipo principal para esse objecto.

Estas duas razões levam a duas propriedades inerentes à própria definição de tipos principais, dada uma operação sintáctica O entre tipos:

1. Existência de tipos principais completos, isto é:

$$\exists_\pi(\vdash M : \pi \wedge (\forall_\tau[\vdash M : \tau \Rightarrow \tau = O(\pi)]))$$

2. Existência de tipos principais íntegros:

$$\exists_\pi(\vdash M : \pi \wedge (\forall_\tau[\tau = O(\pi) \Rightarrow \vdash M : \tau])))$$

Embora a completitude seja exigida e respeitada por todos os sistemas de tipos com a propriedade do tipo principal, o mesmo não acontece com a integridade. Como vamos ver existem extensões ao sistema original de Curry onde os tipos principais, quando existem, não são íntegros, isto é, podemos ter tipos que derivamos do tipo principal de um determinado objecto e que não tipificam esse objecto. Concretamente, estamos a referir-nos aos sistemas de tipos com intersecções. Estes sistemas surgem como uma extensão ao sistema original de Curry, na medida em que tipificam mais termos do que o sistema original, e como uma alternativa ao polimorfismo paramétrico.

Hindley definiu o primeiro sistema polimórfico, [Hin69]. Posteriormente este sistema foi revisto e acrescentado por Milner [Mil40]. Damas expandiu o sistema anterior, [DM82], num sistema que originou o sistema de tipos da linguagem de programação ML. Neste sistema, surge o conceito de esquema de tipos como representação do conjunto de tipos admissíveis para cada expressão. Um esquema de tipos não é mais do que um tipo polimórfico em que os parâmetros são variáveis nos tipos quantificadas universalmente. Neste sistema, é introduzida uma regra que explicitamente garante que, se um termo M tem um tipo polimórfico $\forall_\alpha.\sigma$, então, qualquer tipo obtido pela substituição de α por qualquer tipo em σ também tipifica M. Esta é a chamada regra de instância que preserva uma propriedade do sistema original de Curry que é a seguinte:

$$\vdash M : \sigma \wedge \tau = S(\sigma) \Rightarrow \vdash M : \tau$$

onde S é a operação substituição.

Esta propriedade a que nós chamamos integridade da noção de instância é fundamental em qualquer sistema de tipos descritivo que modele alguma forma de polimorfismo.

Voltando à motivação original de Curry, um tipo polimórfico não é mais do que a representação de um conjunto de descrições de um objecto. No caso do polimorfismo paramétrico, a definição desse conjunto é conseguida através de uma operação (no caso do λ-calculus e do ML a substituição) que, partindo do tipo polimórfico, permite chegar a todas as descrições que ele representa. É, portanto, imperativo que todas as descrições representadas por um tipo polimórfico de um objecto sejam descrições desse mesmo objecto. A integridade da instância, além de ser por si só exigível a qualquer sistema de tipos, implica trivialmente a integridade do tipo principal, e, como vamos ver, a sua falta origina diversos problemas.

Quando iniciámos o estudo de sistemas de tipos com intersecções, verificámos que este é apresentado como uma alternativa ao polimorfismo paramétrico de Milner. No sistema de Damas-Milner, é acrescentada uma nova expressão ao λ-calculus da forma $let\ x = M_1 in\ M_2$. Esta expressão, embora semelhante ao β-redex $(\lambda x.M_2)M_1$, permite tipificar termos que, de outro modo, não têm tipo admissível. Acontece que, se usarmos uma variável x polimorficamente numa expressão M, a abstracção $\lambda x.M$ pode não ser tipificável. Como exemplo considere-se o termo $\lambda x.xx$ que, quando aplicado a $\lambda y.y$, não admite tipo. No sistema de Damas-Milner, consegue-se inferir um tipo para $let\ x = \lambda y.y\ in\ xx$, pois, aqui, x pode ser usado polimorficamente desde que o mesmo tipo polimórfico seja inferível para $\lambda y.y$.

O sistema de tipos com intersecções resolve este problema tipificando $\lambda x.M$ quando x é usado polimorficamente em M. Neste sistema $\lambda x.xx$ tem tipo $(\alpha \cap \alpha \to \beta) \to \beta$. O polimorfismo de x é aqui definido explicitamente afirmando que x pode ser do tipo α e $\alpha \to \beta$. Ao aplicar $\lambda x.xx$ a qualquer termo M, exige-se que M seja tipificável por α e $\alpha \to \beta$ para obter $\vdash (\lambda x.xx)M : \beta$.

Quando iniciámos o estudo de sistemas de tipos com intersecções, começámos por investigar a existência de tipos principais para este sistema. Em [Cop80] é referido que uma extensão à demonstração apresentada em [CDV80] da existência de tipos principais para uma extensão ao sistema original demonstra que o sistema original também tem a propriedade do tipo principal. Isso não é verdade, pois a demonstração apresentada em [CDV80] é baseda numa propriedade, o fecho por igualdade β, que o sistema original não tem. Mais pormenores sobre este assunto são apresentados no terceiro capítulo.

Numa tentativa de usar outro método de demonstração, deparámos com o seguinte: a operação que permite chegar a qualquer tipo a partir do tipo principal, ou seja, a noção de instância, não pode aqui ser definida somente à custa da substituição. Esta operação é constituida pela composição da substituição com outra operação, denominada originalmente por Coppo e Dezanni por expansão e posteriormente dividida em sub-operações, a expansão propriemente dita e a elevação. Para dar uma breve ideia da necessidade de expansões, considere-se o exemplo seguinte:

Considere-se a relação $\{y : \beta\} \vdash \lambda x.y : \alpha \to \beta$, com o significado de que se y tiver tipo β então $\lambda x.y$ tem tipo $\alpha \to \beta$. Isto é verdade, pois $\{y : \beta, x : \alpha\} \vdash y : \beta$. Mas no sistema de tipos com intersecções de $x : \sigma_1 \cap ... \cap \sigma_n$ podemos inferir $x : \sigma_i$

para $1 \leq i \leq n$, logo $\{y : \beta\} \vdash \lambda x.y : \sigma_1 \cap ... \cap \sigma_n \to \beta$ para qualquer $n > 0$. Não existe nenhuma susbstituição de variáveis de tipos por tipos que transforme $\alpha \to \beta$ em $\sigma_1 \cap ... \cap \sigma_n \to \beta$ e o tipo principal de $\lambda x.y$, se existir, é igual a $\alpha \to \beta$. Por isso, definiram-se operações que expandem a variável α na intersecção com n novas variáveis $\alpha_1 \cap ... \cap \alpha_n$. Esta operação, bastante complexa, é o único modo possível de chegar a todos os tipos admissíveis para um determinado termo a partir do tipo principal desse termo, se existir.

Um estudo mais aprofundado de propriedades da expansão leva-nos a algumas surpresas. Consideremos o seguinte exemplo: consideremos o tipo $(\alpha \to \beta) \to (\alpha \to \beta)$ para a função identidade. Este tipo tipifica essa função. Nos sistemas de tipos com intersecções, como vimos anteriormente, o tipo $(\alpha \to \beta) \to (\alpha \cap \alpha_1) \to \beta$ é uma instância de $(\alpha \to \beta) \to (\alpha \to \beta)$. É fácil verificar que $(\alpha \to \beta) \to (\alpha \cap \alpha_1) \to \beta$ não tipifica a função identidade. Este exemplo mostra que podemos ter um tipo para um determinado termo cujas instâncias não tipificam o termo, ou seja, a noção de instância não é íntegra. Como vamos ver no terceiro capítulo, o problema não está na operação de instância, pois esta tem que estar necessariamente definida exactamente como está, mas sim no sistema de tipos. Acontece que, tal como no polimorfismo paramétrico, temos que acrescentar uma regra ao sistema de tipos que explicitamente permita derivar as instâncias de qualquer tipo de uma expressão para essa expressão. Demonstrámos que basta exigir explicitamente que a instância de um tipo tipifique os mesmos termos tipificados pelo tipo mais geral para as variáveis. Ao acrescentar essa regra ao sistema de tipos de Coppo-Dezani obtemos um novo sistema de tipos com intersecções. Salientamos o facto da construção desta extensão ao sistema original ser justificada por uma falha nesse mesmo sistema. Demonstrámos que esse novo sistema mantém a expressividade do original, isto é, tipifica exactamente os termos fortemente normalizáveis, e tem a propriedade do tipo principal usando uma noção de instância íntegra e que, portanto, origina tipos principais íntegros e completos. O trabalho apresentado no terceiro capítulo salienta o facto de como uma extensão ad-hoc do sistema original de Curry, sem obedecer a propriedades fundamentais de sistemas de tipos descritivos, pode originar problemas.

1.4 Programação em Lógica

Uma propriedade semelhante à integridade forte vai assumir particular importância no paradigma da programação em lógica. Neste paradigma, há duas técnicas fundamentais para inferir tipos descritivos. A primeira consiste em inferir um conjunto de regras cuja semântica é uma aproximação da semântica do programa. A segunda aproxima directamente a semântica do programa. Esta aproximação pode ser da semântica operacional, por exemplo a interpretação abstracta top-down, ou da semântica declarativa, como é o caso da interpretação abstracta bottom-up. Em qualquer dos casos, a noção de programa P bem tipificado por um tipo τ é a de que a semântica de τ é uma aproximação conservativa da semântica de P, isto é, $\|P\| \subseteq \|\tau\|$, o que é equivalente à integridade fraca.

Levado ao extremo, a base de Herbrand de um determinado programa P, que é constituida por todos os átomos básicos que podem ser construidos a partir dos símbolos existentes no programa, tipifica P. A exigência da integridade forte leva a considerar como tipos admissíveis apenas as melhores descrições da semântica de P, o que nós denominámos como tipos próprios. Uma aplicação deste modo de caracterizar tipos é apresentado no decorrer da tese tendo em vista a escolha de boas linguagens de tipos para programação em lógica. Nesse capítulo, demonstramos que, para uma classe de tipos usada em programação em lógica e muitas vezes abandonada por razões pragmáticas, como a impossibilidade de obtenção de unificadores mais gerais entre dois tipos, nem sempre é possível obter tipos próprios para um determinado programa. Em contrapartida, para outra classe, usada com algum sucesso para inferência de tipos neste paradigma, demonstrámos a existência de tipos próprios para todos os programas. Estes exemplos mostram que, na definição de sistemas de tipos, a integridade forte é uma condição necessária que devemos respeitar se quisermos obter tipos descritivos que forneçam uma informação útil sobre os objectos.

1.5 Apresentação da Tese

Propomo-nos com este trabalho contribuir para melhorar o grau de conhecimento do que, a nível semântico, impõe certas escolhas subjacentes à definição de sistemas de tipos descritivos para linguagens declarativas. Os nossos objectivos principais são:

1. Definir, sob o ponto de vista semântico, a relação entre tipos e termos que garantem que os primeiros são descrições úteis e íntegras com respeito à motivação inicial dos sistemas;

2. Justificar as más propriedades de alguns sistemas de tipos, nomeadamente, sistemas de tipos com intersecções, usando a metodologia desenvolvida para

dar solução ao ponto 1;

3. Usar uma abordagem semântica para unificar e identificar propriedades de sistemas de tipos no paradigma da programação em lógica, dada a proliferação de metodologias para a definição de sistemas neste paradigma.

A estrutura da continuação da tese é a seguinte:

Capítulo 2: O Sistema de Curry

Neste capítulo, fazemos uma breve descrição do λ-calculus e do sistema de Curry. Esta descrição é fundamental para a compreensão da tese.

Capítulo 3: O sistema de Damas-Milner

Neste capítulo, continuamos a abordar conceitos fundamentais à posterior apresentação de ideias. Neste caso, fazemos a descrição do sistema de Damas-Milner e de algumas propriedades básicas do polimorfismo paramétrico.

Capítulo 4: Sistemas de Tipos com Intersecções

Aqui é feita uma apresentação dos principais sistemas de tipos com intersecções existentes. A descrição dos sistemas é feita atendendo aos problemas existentes em cada um deles e ao modo como outros sistemas são definidos, de modo a resolver esses problemas.

Capítulo 5: Sistemas de Tipos com Restrições

Os resultados básicos da tese são apresentados neste capítulo. Identificamos uma relação semântica entre tipos e termos imposta pela própria relação semântica original $\models$. Essa nova relação é expressa usando conjuntos de inequações nos tipos. A motivação original do sistema de Curry aparece como a escolha de um subconjunto de soluções desse conjunto de inequações (ou restrições), o que fornece uma caracterização semântica do próprio sistema de Curry.

Capítulo 6: Polimorfismo

A metodologia desenvolvida no capítulo anterior é aplicada neste capítulo à caracterização semântica de sistemas polimórficos. Demonstramos que o sistema de tipos da linguagem ML é íntegro com respeito a essa caracterização semântica.

Capítulo 7: Integridade dos Sistemas de Tipos com Intersecções

Neste capítulo, é abordada a falta de integridade da noção de instância no sistema de Coppo-Dezanni. Definimos um novo sistema de tipos de modo a resolver este problema e demonstramos que tem a propriedade do tipo principal e a mesma expressividade do sistema inicial. Finalmente, justificamos a falta de integridade da noção de instância com a falta de integridade do sistema de Coppo-Dezanni com respeito à relação semântica entre tipos e termos obtida aplicando a mesma metodologia dos dois capítulos anteriores.

Capítulo 8: Tipos em Programação em Lógica

Aqui, apresentamos o paradigma da programação em lógica e as metodologias mais importantes na definição de sistemas de tipos neste paradigma.

Capítulo 9: Estudo Semântico de Tipos em Programação em Lógica

Neste capítulo, apresentamos uma caracterização semântica da noção de programa bem tipado e usamos essa caracterização no estudo e levantamento de propriedades das duas linguagens de tipos mais usadas neste paradigma: a linguagem regular e a linguagem regular determinística.

Capítulo 10: Conclusões

Neste capítulo, concluimos e indicamos linhas de investigação resultantes deste trabalho para trabalhos futuros.

Capítulo 2

O sistema de Curry

2.1 Introdução

Universos não tipificados, como o λ-calculus, a teoria de conjuntos, ou na área de linguagens de programação, o PROLOG ou o LISP puro, têm a elegância inerente a sistemas com uma grande flexibilidade de aplicação. Embora a semântica de tais universos seja, de um modo geral, muito simples e os objectos semânticos representados sejam únicos (funções no λ-calculus, conjuntos na teoria de conjuntos ou termos no PROLOG) eles são usados para representar objectos de naturezas semânticas diversas e bem definidas. Por exemplo $\lambda xy.x$ e $\lambda xy.y$ são usados no λ-calculus para representar valores booleanos. Além disso, certas construções sintácticas só fazem sentido quando aplicadas a objectos de um determinado tipo. Utilizando o exemplo anterior *if* B *then* P *else* Q é representado no λ-calculus por BPQ, mas BPQ só tem a semântica do *if then else*, se B for um valor boleano.

O conceito de *tipo* aparece aqui bem definido como sendo um subconjunto do universo semântico.

Os tipos podem ser usados para restringir o uso de objectos. Por exemplo, considerando a função identidade no λ-calculus $\lambda x.x$ e afirmando que tem tipo $int \rightarrow int$, podemos estar a restringir a função aos números inteiros. Outra visão dos tipos é a de propriedades dos objectos. Nesta perspectiva, o tipo $int \rightarrow int$ para $\lambda x.x$ significa que a identidade é uma função, que, quando aplicada a um número inteiro, fornece um inteiro. Estes dois modos de interpretar tipos correspondem no λ-calculus à visão de Church, em que os termos são versões tipificadas dos termos do λ-calculus e à visão de Curry, em que os termos pertencem a um universo não tipificado e os tipos são vistos como propriedades de objectos não tipificados. Nas linguagens de programação, existem duas noções equivalentes: a tipagem explícita e implícita. No primeiro caso, incluem-se linguagens como o PASCAL, em que os programas são escritos juntamente com as declarações de tipos e o compilador executa uma verificação da consistência dessas declarações. No segundo caso, incluem-se linguagens como o ML, em que o programa é escrito sem declarações de tipos e estes são inferidos pelo compilador. Estas duas versões de tipos são também conhecidas como visão prescritiva - a tipagem explícita, e descritiva - tipagem implícita.

No decorrer desta tese, assumimos uma visão descritiva dos tipos. Todos os resultados apresentados partem do princípio que temos universos não tipificados em que os objectos se podem agrupar de acordo com propriedades comuns - os tipos.

Tendo como objectivo obter uma caracterização semântica de propriedades conhecidas de sistemas de tipos, vamos começar por estudar sistemas de tipos simples para o λ-calculus, também conhecido por sistema de Curry. A escolha do λ-calculus como objecto inicial do nosso estudo foi baseada em três razões fundamentais: o facto de ser a base do paradigma da programação funcional, o facto de possuir uma semântica denotacional que usa domínios semânticos idênticos aos domínios usados por grande parte das linguagens de programação e por ser uma área onde existe um estudo mais aprofundado de sistemas de tipos para linguagens de programação. Neste capítulo, começamos por apresentar a noção de tipo de um modo semântico, isto é, como um conjunto de valores com determinadas características. A noção semântica de tipagem surge naturalmente como: um termo M tem um tipo τ, representado por $M : \tau$ se o valor denotado por M pertence à semântica de τ. Apresentamos depois o sistema original de Curry para o λ-calculus. Neste sistema, é notório que nem todos os tipos para um determinado termo (válidos sob o ponto de vista semântico) são deriváveis para esse termo. O conjunto de termos para os quais é possível derivar um tipo no sistema de Curry vai ser caracterizado no capítulo 5.

2.2 O λ-calculus

O λ-calculus criado por Church, [Chu33], revelou-se um modelo das funções computáveis e deu origem ao paradigma da programação funcional. O nosso objectivo não é dar uma explicação exaustiva do λ-calculus, mas só apresentar as bases necessárias para o estudo posterior sobre teoria de tipos. Em particular, vamos ver, com algum pormenor, a semântica denotacional de Scott-Strachey [Sto77].

2.2.1 Funções

Partindo de um domínio semântico de valores V, o λ-calculus é um modelo para as funções de V em V. Sendo assim, temos neste modelo duas operações:

1. A aplicação, ou seja, aplicar uma função F a um argumento A, FA. Sob o ponto de vista computacional, F é vista como um programa e A como o input de F. Repare-se que A pode ser também uma função, ou seja, o nível semântico de V e de $V \to V$ é o mesmo, o que, como vamos ver, condiciona a escolha das estruturas usadas como domínio semântico para o λ-calculus.

2. A abstracção que permite representar funções de um parâmetro x. Esta operação é representada por $\lambda x.M$ e é um modelo para a função:

$$\begin{aligned} f: \quad & V \to V \\ & x \to M \end{aligned}$$

Se aplicarmos $\lambda x.M$ a outro termo N, $(\lambda x.M)N$, o resultado é a substituição de x por N em M.

Exemplo 2.1 *Abusando da notação, este exemplo pretende clarificar o uso da aplicação e da abstracção:*

$$(\lambda x.x + 2)(5) = 5 + 2 = 7$$

A redução $(\lambda x.M)N = M[x/N]$ é chamada conversão β e é o axioma básico da teoria.

2.2.2 Sintaxe

O conjunto de termos, Λ, é definido do seguinte modo: Partindo de um conjunto infinito de variáveis Var e sendo $x \in Var$ e M e M' termos temos:

$$M ::= x|MM'|\lambda x.M$$

2.2.3 Conjuntos Parcialmente Ordenados Completos

Como foi referido no λ-calculus, os domínios semânticos dos valores V e das funções de V em V estão ao mesmo nível, isto é, podemos ter o caso de uma função ser aplicada a ela própria. Isto faz com que qualquer domínio semântico tenha que ser tal que exista um isomorfismo entre V e $V \rightarrow V$. Os conjuntos parcialmente ordenados completos têm a propriedade anterior e são utilizados como domínios semânticos para o λ-calculus. Por esta razão, e antes de apresentar a semântica escolhida para o λ-calculus, vamos apresentar uma breve introdução ao tema.

Definição 2.1 *Um conjunto parcialmente ordenado D é um conjunto D onde está definida uma relação de ordem parcial $\sqsubseteq$.*

Definição 2.2 *Um conjunto parcialmente ordenado D é completo se tiver elemento mínimo $\perp_B$ e se toda a cadeia ascendente tiver um supremo.*

A partir de agora, usamos a sigla CPO (Complete Partial Order) para conjunto parcialmente ordenado completo.

Dados dois CPOs D e E, diz-se que uma função $f : D \rightarrow E$ é monótona se $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(Y)$. Seja $\bigsqcup M$ o supremo de uma cadeia ascendente M. A função f é contínua se é monótona e para qualquer cadeia ascendente M, $f(\bigsqcup M) = \bigsqcup f(M)$. Sejam f e g duas funções do CPO D no CPO E. Então $f \sqsubseteq g$ se e só se $f(x) \sqsubseteq g(x) \forall_{x \in D}$. Com esta ordem, o conjunto parcialmente ordenado das funções contínuas entre dois CPOs é um CPO. Dados dois CPOs, D e E, o produto $D \times E$ é definido por:

$$D \times E = \{(x, y) | x \in D \wedge y \in E\}$$

Definindo $\bigsqcup(D \times E) = (\bigsqcup D, \bigsqcup E)$ e $\perp_{D \times E} = (\perp_D, \perp_E)$ $D \times E$ é também um CPO. Vamos agora definir a soma amalgamada entre dois CPOs E e D, $E \oplus D$:

$$D \oplus E = ((D - \{\perp_D\} \times \{0\}) \cup ((E - \{\perp_E\} \times \{1\}) \cup \{\perp_{D \oplus E}\}$$

. Consideremos a seguinte ordem parcial em $E \oplus D$:

1. $\perp_{D \oplus E} \sqsubseteq z, \forall_{z \in D \oplus E}$
2. $(x, m) \sqsubseteq (y, n)$ se e só se $m = n$ e $x \sqsubseteq y$

$D \oplus E$ ordenado pela relação anterior é um CPO. As duas funções injectivas de D e E em $D \oplus E$ são definidas como:

$$in_D(x) = \begin{cases} (x, 0) & \text{se } x \neq \perp \\ \perp_{D \oplus E} & \text{se } x = \perp \end{cases}$$

$$in_E(x) = \begin{cases} (x, 1) & \text{se } x \neq \perp \\ \perp_{D \oplus E} & \text{se } x = \perp \end{cases}$$

As definições anteriores são extensíveis à soma amalgamada de n CPOs $D_1 \oplus ... \oplus D_n$. Finalmente, se um CPO D for uma parcela de um CPO soma V, definimos a função $v_{|D} : V \to D$ como:

$$v_{|D} = \begin{cases} d & \text{se } \exists_{d \in D} . v = in_D(d); \\ \perp_D & \text{caso contrário} \end{cases}$$

2.2.4 Semântica

Estamos agora em condições de definir uma semântica para o λ-calculus. Como vimos, dado um CPO D o conjunto de funções contínuas de D em D, $D \to D$ é um CPO. Dado um CPO de valores B e um CPO de funções F, o domínio semântico é definido pelas equações:

$$\begin{aligned} V &= B \oplus F \\ F &= V \to V \end{aligned}$$

A existência de soluções para as equações anteriores é baseada em resultados conhecidos ([Plo76]). A semântica das variáveis é dada através de um ambiente: uma função do conjunto de variáveis nos valores. O domínio dos ambientes Env é o CPO $Env = Var \to V$. Dado um ambiente ρ, um valor v e uma variável x, definimos o ambiente $\rho[v/x]$ como:

$$\rho[v/x](y) = \begin{cases} v & \text{se } x = y \\ \rho(y) & \text{caso contrário} \end{cases}$$

A função que dá a semântica do λ-calculus, é dada por:

$$E : \Lambda \to Env \to V$$

$$\begin{array}{lcl} E\|x\|\rho & = & \rho(x) \\ E\|e_1e_2\|\rho & = & ((E\|e_1\|\rho)_{|F})(E\|e_2\|\rho) \\ E\|\lambda x.e\|\rho & = & in_F(\lambda.vE\|e\|\rho[v/x]) \end{array}$$

2.3 Tipos

Assumimos que temos um conjunto de variáveis de tipos T_V. Partindo do conjunto T_V, temos depois tipos que representam funções que, quando aplicadas a elementos de um tipo α, têm como resultado um elemento do tipo β, representados por $\alpha \rightarrow \beta$. Finalmente, dado um conjunto T_V de variáveis, a sintaxe da linguagem de tipos é dada por :

$$\tau ::= \sigma | \tau \rightarrow \tau'$$

em que σ é uma variável nos tipos e τ e τ' são tipos. O domínio semântico dos tipos é constituido por subconjuntos do domínio semântico do λ-calculus V, chamados ideais:

Definição 2.3 *Um subconjunto não vazio I de um CPO V é um ideal se e só se:*

- *Se $v \in I$ e $v' \sqsubseteq v$ então $v' \in I$;*
- *O supremo de qualquer cadeia ascendente em I também pertence a I.*

Definimos o domínio semântico dos tipos como o conjunto $\overline{V}$ de todos os ideais de V. Vamos assumir que existe uma função ε que dá a semântica das variáveis de tipos. A semântica dos tipos é dada pela função $T : Tipos \rightarrow \overline{V}$ definida por:

$$\begin{array}{lcl} T\|\sigma\|\varepsilon & = & \varepsilon(\sigma) \\ T\|\tau_1 \rightarrow \tau_2\|\varepsilon & = & \{in_F(f) | f \in V \rightarrow V : f(T\|\tau_1\|\varepsilon) \subseteq T\|\tau_2\|\varepsilon\} \end{array}$$

Estamos agora em condições de formalizar o conceito *ter um determinado tipo.* Vamos definir uma relação entre pares $< variável, tipo > A$, termos M e tipos τ, com o significado: *assumindo a declaração A para cada variável, o termo M tem tipo τ.* A noção de declaração de tipo é definida por:

Definição 2.4 *Uma declaração é um par da forma $x : \tau$, onde x é uma variável do λ-calculus e τ é um tipo. Uma declaração é uma relação unária em que o tipo é o predicado e a variável o sujeito.*

Definição 2.5 *Dado um ambiente ρ e uma função ε que dá a semântica das variáveis de tipos, dizemos que A é uma base válida, representada por $\rho :_\varepsilon A$ se e só se $\forall_{x:\tau \in A}.\rho(x) \in T\|\tau\|\varepsilon$. Por vezes, para não sobrecarregar a notação, escrevemos $\rho : A$ com o significado de $\rho :_\varepsilon A$.*

A relação $\rho : A$ formaliza a noção intuitiva de que τ é válida para uma variável x, se a semântica de x pertence à semântica de τ. Finalmente, definimos a relação $A \models M : \tau$, com o significado: *Para todo o ambiente ρ, se A é uma base válida ,então, o λ-termo M tem tipo τ*

Definição 2.6 *Dada uma base A, um termo M e um tipo τ:*

$$A \models e : \tau \Leftrightarrow \forall_\varepsilon \forall_{\rho \in Env}[\rho :_\varepsilon A \Rightarrow E\|e\|\rho \in T\|\tau\|\varepsilon]$$

Para realçar o carácter semântico da relação $\models$, temos o exemplo muito simples de que tipos são válidos para uma variável x para a qual temos a declaração $x : int$. Vamos assumir que existem tipos básicos int e $real$ (inteiros e reais), e que $T\|int\| \subseteq T\|real\|$. Então $\{x : int\} \models x : int$ pois $\forall_{\rho \in Env}.\rho(x) \in T\|int\| \Rightarrow \rho(x) \in T\|int\|$. Mas também temos $\{x : int\} \models x : real$ pois $\forall_{\rho \in Env}.\rho(x) \in T\|int\| \Rightarrow \rho(x) \in T\|real\|$. De um modo geral, $\{x : \tau_1\} \models x : \tau_2$ para todo o τ_2 tal que $T\|\tau_1\| \subseteq T\|\tau_2\|$. Isto corresponde à ideia intuitiva de que, se declararmos um tipo τ_1 para x, então, x tem a propriedade τ_1 e também tem qualquer propriedade que inclua τ_1.

2.4 O Sistema de Tipos Simples - Sistema de Curry

Sistemas de tipos para linguagens de programação ou λ-calculus explicitam que tipos podemos derivar para um dado objecto. Esta derivação tem um carácter sintáctico semelhante à noção de derivação em lógica.

A partir de uma base, existem regras de derivação que definem operações sintácticas válidas com vista à derivação de um tipo para um determinado objecto. Dizemos que o tipo τ é derivável para M a partir de A, representado por $A \vdash M : \tau$, se $A \vdash M : \tau$ pode ser obtido pela aplicação de um conjunto de regras de inferência pertencentes ao sistema de tipos. Em 1934 foi apresentado o primeiro sistema de tipos descritivo, [Cur34]. Este sistema deu origem ao sistema de tipos simples para o λ-calculus e é constituido por três regras de inferência:

- $A \vdash x : \tau$ se $(x : \tau) \in A$
- $A \vdash MN : \tau$ se $A \vdash M : \sigma \to \tau$ e $A \vdash N : \sigma$
- $A \vdash \lambda x.M : \sigma \to \tau$ se $A \cup \{x : \sigma\} \vdash M : \tau$

Estas regras são conhecidas como o *Axioma*, a *Aplicação ou* $\to$*-Eliminação* e a *Abstracção ou* $\to$*-Introdução.*

As regras anteriores aparecem, normalmente, na seguinte notação:

$$AXIOMA \quad A \vdash x : \tau \text{ ,se } x : \tau \in A$$

$$APL. \qquad \frac{A \vdash M : \sigma \to \tau, A \vdash N : \sigma}{A \vdash MN : \tau}$$

$$ABS. \qquad \frac{A \cup \{x : \sigma\} \vdash M : \tau}{A \vdash \lambda x.M : \sigma \to \tau}$$

Uma derivação pode ser representada por uma árvore em que cada nó é obtido dos nós imediatamente acima dele, por uma regra de derivação.

Exemplo 2.2 *Neste exemplo é representada a árvore de derivação para* $\vdash (\lambda x.x)(\lambda x.x) : \alpha \to \alpha$*:*

$$\begin{tikzcd}
\{x : \alpha \to \alpha\} \vdash x : \alpha \to \alpha \arrow[d, "ABS"'] & & \{x : \alpha\} \vdash x : \alpha \arrow[d, "ABS"'] \\
\vdash \lambda x.x : (\alpha \to \alpha) \to (\alpha \to \alpha) \arrow[dr] & & \vdash \lambda x.x : \alpha \to \alpha \arrow[dl, "APL"'] \\
& \vdash (\lambda x.x)(\lambda x.x) : \alpha \to \alpha &
\end{tikzcd}$$

Uma propriedade fundamental dos sistemas de tipos é a integridade, isto é, se um tipo τ é derivado para um termo M a partir de uma base A, então, a semântica de M é um valor de tipo τ. Esta condição traduz-se por

$$A \vdash M : \tau \Rightarrow A \models M : \tau$$

O oposto não se verifica. $A \models M : \tau \not\Rightarrow A \vdash M : \tau$. O facto de a derivação não ser completa, isto é, haver tipos semânticamente válidos que não são deriváveis para um dado termo, deve-se fundamentalmente a duas razões:

1. Inferência de tipos. Ligado ao conceito de derivação de tipos surge a inferência de tipos, isto é, dada uma base A e um termo M, conseguimos ou não ter um algoritmo capaz de fornecer o tipo de M. Em geral, tal algoritmo não existe e a solução é limitar o conjunto de termos para os quais é possível derivar um tipo. Assim sendo, podemos ter tipos válidos sob o ponto de vista semântico que não podem ser derivados. Por exemplo $\models (\lambda x.xx)(\lambda y.y) : \alpha \to \alpha$ e no sistema de Curry não é possível derivar um tipo para $(\lambda x.xx)(\lambda y.y)$

2. Para um dado termo M, há tipos semânticos que são gerais demais. Não nos interessa derivar esses tipos. Um exemplo é $\models \lambda x.x : \alpha \to \beta$, desde que $T\|\alpha\| \subseteq T\|\beta\|$, e $\not\vdash \lambda x.x : \alpha \to \beta$.

2.5 Substituições

As variáveis de tipo são objectos sintácticos que podem ser substituidos por tipos. Neste contexto, as noções de substituição e instância surgem naturalmente como:

Definição 2.7 *Uma substituição $S = [\tau_1/\alpha_1, ..., \tau_n/\alpha_n]$ é uma função do conjunto de variáveis de tipos no conjunto de tipos que transforma cada α_i em τ_i e todas as outras variáveis nelas próprias.*

Esta noção é extensível a uma função do conjunto de tipos nele próprio, substituindo cada ocorrência de uma variável α num dado tipo por $S\alpha$.

Definição 2.8 *Um tipo σ' é uma instância de um tipo σ se e só se existe uma substituição S tal que $\sigma' = S\sigma$.*

2.6 Tipos Principais

No estudo de sistemas de tipos descritivos, uma propriedade assume particular importância: o facto de, dado um termo M, existir um tipo de M mais geral do que todos os outros. Geral, aqui, surge no sentido :τ_1 *é mais geral do que* τ_2 *se* τ_2 *for uma instância de* τ_1. Esta propriedade é conhecida por *propriedade do tipo principal* e verifica-se para o sistema de Curry, [Hin69].

Definição 2.9 *Dado um termo* M*, o par* $< P, \pi >$ *é o par principal de* M*, respectivamente <Base pricipal, tipo principal>, se e só se:*

1. $P \vdash M : \pi$
2. *Para qualquer derivação* $B \vdash M : \sigma$ *há um subconjunto* B' *de* B *tal que* B' *é uma instância de* P *e* σ *é uma instância de* π.

A propriedade do tipo principal assume particular importância quando é necessário ter inferência de tipos. Neste caso para um dado termo M, basta inferir o tipo principal, para obtermos todos os tipos derivaveis para M. No caso de posteriormente necessitarmos de fazer verificação de tipos, basta verificar que se trata de uma instância do tipo principal.

Partindo de uma relação $\sqsubseteq$ entre tipos com o significado $\tau_1 \sqsubseteq \tau_2$ se τ_1 for mais geral do que τ_2, o tipo principal é a intersecção de todos os tipos deriváveis para um dado termo:

Definição 2.10 *Dado um termo* M*,* $< P, \pi >$ *é o par principal de* M *se e só se:*

1. $P \vdash M : \pi$
2. $< P, \pi > = \bigcap \{< B, \tau > | B \vdash M : \tau\}$

2.7 Inferência de Tipos

O sistema de Curry é decidível. Existe um algoritmo, baseado no algoritmo de unificação de Robinson, [Rob65], que dados uma base A e um termo M retorna o tipo principal de M. O algoritmo usado é o seguinte:

Definição 2.11 *Seja* A *uma base,* M *um termo e* τ *um tipo. Seja* $UNIFY$ *a função que define o algoritmo de unificação de Robinson tal que* $UNIFY(\tau_1, \tau_2) = S$ *, onde* S *é uma substituição tal que* $S\tau_1 = S\tau_2$*. O algoritmo de inferência de tipos de Curry para o lambda calculus é definido pela função* $T(M) = (A, \tau)$*, onde:*

1. *Se M é uma variável x então $A = \{x : \alpha\}$ e $\tau = \alpha$, onde α é uma variável nova;*

2. *Se $M \equiv M_1 M_2$, $T(M_1) = (A_1, \tau_1)$ e $T(M_2) = (A_2, \tau_2)$, então $T(M) = (S(A_1 \cup A_2), S\alpha)$, onde $S = UNIFY(\tau_1, \tau_2 \to \alpha)$ e α é uma variável nova;*

3. *Se $M \equiv \lambda x.N$ e $T(N) = (A_N, \sigma)$ então:*

   (a) *Se não há hipóteses para x em A_N, $T(M) = (A_N, \alpha \to \sigma)$, onde α é uma variável nova;*

   (b) *Se há uma hipótese $\{x : \tau\} \in A_N$, $T(M) = (A_N - \{x : \tau\}, \tau \to \sigma)$.*

2.8 Discussão

Neste capítulo, apresentámos o primeiro sistema de tipos descritivos. Apresentámos uma semântica para termos baseada em conjuntos parcialmente ordenados. Os tipos surgem como conjuntos parcialmente ordenados com determinadas características: os ideais. Em seguida, foi apresentada a noção de tipificação semântica e demonstrada a integridade do sistema.

No capítulo seguinte, vamos apresentar a extensão ao sistema de Curry para lidar com polimorfismo que deu origem à linguagem de programação ML. Nos capítulos 5 e 6, vamos definir sistemas de tipos baseados directamente na relação semântica entre termos e tipos apresentadas aqui e vamos ver como se relacionam com os sistemas tradicionais apresentados neste e no próximo capítulo.

Capítulo 3

O Sistema de Damas-Milner

3.1 Introdução

O sistema de Curry tem a propriedade de poder tipificar um termo com mais do que um tipo. Partindo do princípio que podemos acrescentar descrições de conjuntos de tipos (esquemas de tipos) à própria linguagem de tipos, podemos aumentar o sistema anterior de modo a permitir a derivação de tipos polimórficos (que representem mais do que um tipo). O sistema paradigmático que lida com polimorfismo é o sistema de Damas-Milner, e este capítulo faz uma descrição desse sistema.

Começamos com uma descrição da linguagem de termos e uma justificação do porquê acescentar novos termos ao λ-calculus. Passamos depois à descrição dos tipos polimórficos e à apresentação da sua semântica. Finalmente, descrevemos algumas propriedades do sistema.

Tal como o capítulo anterior, também aqui vamos apresentar bases para a compreensão dos próximos capítulos onde vamos fazer um estudo semântico do polimorfismo, qual o seu significado, e que propriedades impõe aos sistemas de tipos.

3.2 O Sistema de Damas-Milner

O sistema de Damas-Milner, [DM82], constitui a base da inferência de tipos na linguagem ML. Este sistema foi construído para uma extensão ao λ-calculus que contém um constructor *let* e originalmente uma representação explícita de pontos fixos, fix.

3.2.1 A linguagem de termos

3.2.1.1 Sintaxe

A sintaxe das expressões é definida por:

$$e ::= x \mid ee' \mid \lambda x.e \mid let\ x = e\ in\ e' \mid fix\ x.e$$

onde x pertence a um conjunto infinito de variáveis. As expressões let e fix.aumentam o conjunto de termos tipificáveis. No β-redex $(\lambda x.e')e$, que é semânticamente equivalente a $let\ x = e\ in\ e'$, quando x é usado polimorficamente em e' a expressão $\lambda x.e'$ pode não ser tipificável. Por exemplo $\vdash (let\ x = \lambda y.y\ in\ xx) : \alpha \to \alpha$ e $(\lambda x.xx)\lambda y.y$ não admite tipo, porque $\lambda x.xx$ não é tipificável neste sistema. O mesmo acontece para a expressão fix. Sob o ponto de vista semântico, $fix\ x.e$ é equivalente a $Y(\lambda x.e)$, onde Y é o operador ponto fixo $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ para o qual não é possível derivar um tipo neste sistema.

3.2.1.2 Semântica

A semântica das expressões let e fix é dada por:

$$\begin{array}{lcl} E\|let\ x = e_1 in\ e_2\|\rho & = & E\|e_2\|\rho[E\|e_1\|\rho/x] \\ E\|fix\ x.e\|\rho & = & \bigcup_i v_i \end{array}$$

onde $v_0 = \perp$ e $v_{i+1} = E\|e\|\rho[v_i/x]$

3.2.2 Esquemas de Tipos

Como objectivo inicial de obter uma representação do conjunto de tipos que é possível inferir para um dado termo a partir de uma determinada base, surge a definição de esquema de tipos. Note-se que dada uma base B, se $B \vdash M : \sigma$ então $B \vdash M : S\sigma$ para todas as substituições em variáveis de σ que não ocorrem em B, pois, neste caso, SB não afecta B. Um esquema de tipos é definido como:

Definição 3.1 *Um esquema de tipos σ é um tipo τ ou um termo da forma $\forall_{\alpha_1,...,\alpha_n}.\tau$, onde τ é um tipo e $\alpha_1, ..., \alpha_n$ são variáveis de tipo chamadas variáveis genéricas.*

As variáveis que não são quantificadas universalmente ocorrem livres. A noção de instância é generalizável a esquemas de tipos por:

Definição 3.2 *Dados um tipo τ e um esquema de tipos σ dizemos que τ é uma instância genérica de σ se e só se $\sigma = \tau$ ou $\sigma = \forall_{\alpha_1,...,\alpha_n}.\eta$ e $\exists_{\tau_1,...,\tau_n}$ tais que $\tau = [\tau_i/\alpha_i]\sigma$*

Usando estas definições, podemos afirmar que, se tivermos $B \vdash M : \sigma$, então, temos $B \vdash M : \tau$ onde τ é qualquer instância genérica de $\forall_{\alpha_1,...,\alpha_n}.\sigma$, sendo $\alpha_1, ..., \alpha_n$ as variáveis de σ que não ocorrem livres em B.

O sistema de Damas e Milner vai mais longe, pois os esquemas de tipos são usados na própria derivação de tipos, incluindo os esquemas de tipos nos tipos que é possível derivar para um termo e declarar para as variáveis que ocorrem livres no termo.

3.2.3 Sintaxe

Neste sistema, os tipos são dados pela seguinte gramática:

$$\begin{array}{rcl} \tau & ::= & \alpha|\tau' \to \tau'' \\ \sigma & ::= & \tau|\forall\alpha.\sigma' \end{array}$$

onde α pertence a um conjunto infinito da variáveis. As expressões de tipos deriváveis de σ são esquemas de tipos. As variáveis quantificadas universalmente são chamadas variáveis genéricas.

3.2.4 Semântica

A semântica dos esquemas de tipos é dada pela intersecção das semânticas de todas as suas instâncias genéricas.

$$T\|\forall_\alpha.\sigma\|\varepsilon = \bigcap_{I \in \bar{V}} T\|\sigma\|\varepsilon[I/\alpha]$$

3.2.5 O Sistema de Tipos

Seja x uma variável, α uma variável de tipo, e e e' expressões, τ e τ' tipos e σ e σ' esquemas de tipos. O sistema de tipos é dado pelas seguintes regras de inferência:

$$AXIOM \qquad A \cup \{x : \sigma\} \vdash x : \sigma$$

$$GEN \qquad \frac{A \vdash e : \sigma \text{ ,} \alpha \text{ não ocorre livre em } A}{A \vdash e : \forall \alpha . \sigma}$$

$$INST \qquad \frac{A \vdash e : \forall \alpha . \sigma}{A \vdash e : \sigma[\tau / \alpha]}$$

$$\rightarrow ELIM \qquad \frac{A \vdash e : \tau' \rightarrow \tau, A \vdash e' : \tau'}{A \vdash (ee') : \tau}$$

$$\rightarrow INTRO \qquad \frac{A \cup \{x : \tau'\} \vdash e : \tau}{A \vdash \lambda x . e : \tau' \rightarrow \tau}$$

$$LET \qquad \frac{A \vdash e : \sigma, A \cup \{x : \sigma\} \vdash e' : \sigma'}{A \vdash \text{let } x = e \text{ in } e' : \sigma'}$$

$$FIX \qquad \frac{A \cup \{x : \tau\} \vdash e : \tau}{A \vdash \text{fix } x . e : \tau}$$

No que diz respeito à derivação de tipos podemos retirar a regra FIX aos sistema se virmos o operador ponto fixo fix $x.M$ como uma variável cujo tipo declarado é $(\alpha \rightarrow \alpha) \rightarrow \alpha$. A versão original do sistema de Damas-Milner foi apresentada desta forma, e a partir de agora, vamos considerar o sistema de Damas-Milner sem a regra de derivação FIX.

3.2.6 Polimorfismo Paramétrico

Em várias linguagens de programação, os procedimentos e funções têm um tipo único. Isto torna essas linguagens pouco ou nada adequadas à elaboração de programas em que se pretende aplicar a mesma função a dados de tipos diferentes. Por exemplo, um simples push ou pop numa pilha de inteiros ou strings tem que ser implementado usando duas funções distintas. O mesmo acontece se quisermos calcular o comprimento de uma lista de inteiros ou de strings. A partir deste último exemplo, verificamos que o tipo da função que retorna o comprimento de uma lista deveria ser uma função de uma lista de elementos arbitrários nos inteiros. Esta noção de *elementos arbitrários* corresponde ao conceito de polimorfismo no sentido de permitirmos ter objectos com mais do que um tipo - com um tipo polimórfico.

Um dos modos de polimorfismo em linguagens de programação é o polimorfismo paramétrico. Neste modo de polimorfismo, os tipos contêm parâmetros que podem ser instanciados por qualquer tipo, permitindo uma representação sintáctica de conjuntos de tipos admissíveis para um determinado objecto. Os esquemas de tipos do sistema de Damas-Milner constituem a base dos sistemas de tipos polimórficos para linguagens de programação. Pegando no exemplo da função *comprimento*, o seu tipo seria o esquema de tipos $comprimento : \forall_{\alpha}.\alpha\ list \rightarrow int$ com o significado óbvio de *para qualquer instância τ da variável genérica α, "lista de τ" $\rightarrow int$.* Este sistema deu origem ao sistema de tipos polimórfico da linguagem de programação ML, [Mil84].

3.2.7 Inferência de Tipos

Antes de descrever o algoritmo de inferência de tipos para o sistema de Damas-Milner vamos definir a noção de fecho de um tipo:

Definição 3.3 *Dado um conjunto de variáveis V e um tipo τ o fecho de τ com respeito a V, $\bar{V}(\tau)$ é o esquema de tipos $\forall_{\alpha_1,\dots,\alpha_n}\tau$, onde $\alpha_1,\dots,\alpha_n$ são todas as variáveis que ocorrem em τ e não pertencem a V. Dada uma base A , $\bar{A}(\tau)$ é o fecho de τ com respeito ao conjunto de variáveis que ocorrem em A.*

O sistema de Damas-Milner é decidível. O algoritmo de inferência de tipos neste sistema é o seguinte:

Definição 3.4 *Seja A uma base, M um termo, S uma substituição e τ um tipo. Seja $UNIFY$ o algoritmo de unificação de Robinson ,[Rob65]. O algoritmo de inferência de tipos para o sistema de Damas-Milner é definido pela função $T(A,M) = (S,\sigma)$, onde:*

1. *Se* M *é uma variável* x *e existe uma declaração* $x : \forall_{\alpha_1,\ldots,\alpha_n}\tau \in A$ *então* S *é a função identidade e* $\sigma = [\beta_i/\alpha_i]\tau$, *onde* β_i *é uma variável nova para* $1 \leq i \leq n$;

2. *Se* $M \equiv M_1 M_2$, *seja:*

   - $T(A, M_1) = (S_1, \tau_1)$;
   - $T(S_1 A, M_2) = (S_2, \tau_2)$;
   - $S_3 = UNIFY(S_2\tau_1, \tau_2 \to \beta)$, *onde* β *é uma variável nova;*

   então $S = S_3 \circ S_2 \circ S_1$ *e* $\sigma = S\beta$;

3. *Se* $M \equiv \lambda x.N$, *seja* β *uma variável nova e* $T(A - \{x : \alpha\} \cup \{x : \beta\}, N) = (S_1, \tau_1)$, *então* $S = S_1$ *e* $\sigma = S_1(\beta \to \tau_1)$;

4. *Se* $M \equiv let\ x = M_1\ in\ M_2$, *seja:*

   - $T(A, M_1) = (S_1, \tau_1)$ *e*
   - $T(S_1(A - \{x : \alpha\} \cup \{x : \overline{S_1 A}(\tau_1)\}), M_2) = (S_2, \tau_2)$.

   então $S = S_2 \circ S_1$ *e* $\sigma = \tau_2$

Em [Dam84] demonstra-se o seguinte teorema:

Teorema 3.1 *Se* $T(A, M) = (S, \sigma)$ *então existe uma derivação no sistema de Damas-Milner de* $SA \vdash M : \sigma$.

3.2.8 Propriedades

O sistema de Damas-Milner tornou-se a base de sistemas tipos para linguagens de programação funcional devido às seguintes propriedades:

1. Permite polimorfismo paramétrico;

2. Possui a propriedade do tipo principal;

3. A inferência de tipos é decidível, isto é, existe um algoritmo que, dado um termo, se ele for tipificável, retorna o tipo principal e falha, caso contrário;

4. O algoritmo de inferência de tipos é íntegro e completo no sentido em que:

(a) Se o algoritmo retorna (S, σ) como tipo de uma expressão E, então $SA \vdash E : \sigma$ e

(b) Se $A \vdash E : \tau$, então o algoritmo retorna (S, σ) como tipo para E e τ é uma instância de σ.

5. O sistema de tipos é íntegro. Isto é $A \vdash E : \sigma \Rightarrow A \models E : \sigma$.

Há vários outros sistemas que modelam polimorfismo paramétrico. Destacamos o sistema de Mycroft-Milner, [Myc84] em que é possível tipificar o ponto fixo polimorficamente. Neste sistema, embora se mantenha a propriedade do tipo principal, perde-se a decidibilidade do sistema.

Nem todos os sistemas de tipos têm a propriedade do tipo principal. Qualquer extensão ao sistema de Damas-Milner que permita alguma forma de polimorfismo nas variáveis quantificadas por λ perde essa propriedade. Um exemplo de um desses sistemas é o sistema F, [Gir72] e [Rey74], em que, além disso, podemos ter esquemas de tipos no lado direito de um predicado $\rightarrow$. Este sistema nem é decidível, [Wel94] nem tem a propriedade do tipo principal, como mostra o próximo exemplo:

1.

$$
\begin{array}{rcl}
\{(f : \forall\alpha.\alpha \rightarrow \alpha), (x : \alpha), (y : \beta)\} & \vdash & f : \alpha \rightarrow \alpha \\
\{(f : \forall\alpha.\alpha \rightarrow \alpha), (x : \alpha), (y : \beta)\} & \vdash & \lambda zw.z : \alpha \rightarrow \beta \rightarrow \alpha \\
\{(f : \forall\alpha.\alpha \rightarrow \alpha), (x : \alpha), (y : \beta)\} & \vdash & fx : \alpha \\
\{(f : \forall\alpha.\alpha \rightarrow \alpha), (x : \alpha), (y : \beta)\} & \vdash & (\lambda zw.z)(fx) : \beta \rightarrow \alpha \\
\{(f : \forall\alpha.\alpha \rightarrow \alpha), (x : \alpha), (y : \beta)\} & \vdash & f : \beta \rightarrow \beta \\
\{(f : \forall\alpha.\alpha \rightarrow \alpha), (x : \alpha), (y : \beta)\} & \vdash & fy : \beta \\
\{(f : \forall\alpha.\alpha \rightarrow \alpha), (x : \alpha), (y : \beta)\} & \vdash & (\lambda zw.z)(fx)(fy) : \alpha \\
\{(x : \alpha), (y : \beta)\} & \vdash & \lambda f.(\lambda zw.z)(fx)(fy) : (\forall\alpha.\alpha \rightarrow \alpha) \rightarrow \alpha
\end{array}
$$

2.

$$
\begin{array}{rcl}
\{(f:\forall\alpha.\alpha\rightarrow\delta),(x:\alpha),(y:\beta)\} & \vdash & f:\alpha\rightarrow\delta \\
\{(f:\forall\alpha.\alpha\rightarrow\delta),(x:\alpha),(y:\beta)\} & \vdash & \lambda zw.z:\delta\rightarrow\delta\rightarrow\delta \\
\{(f:\forall\alpha.\alpha\rightarrow\delta),(x:\alpha),(y:\beta)\} & \vdash & fx:\delta \\
\{(f:\forall\alpha.\alpha\rightarrow\delta),(x:\alpha),(y:\beta)\} & \vdash & (\lambda zw.z)(fx):\delta\rightarrow\delta \\
\{(f:\forall\alpha.\alpha\rightarrow\delta),(x:\alpha),(y:\beta)\} & \vdash & f:\beta\rightarrow\delta \\
\{(f:\forall\alpha.\alpha\rightarrow\delta),(x:\alpha),(y:\beta)\} & \vdash & fy:\delta \\
\{(f:\forall\alpha.\alpha\rightarrow\delta),(x:\alpha),(y:\beta)\} & \vdash & (\lambda zw.z)(fx)(fy):\delta \\
\{(x:\alpha),(y:\beta)\} & \vdash & \lambda f.(\lambda zw.z)(fx)(fy):(\forall\alpha.\alpha\rightarrow\delta)\rightarrow\delta
\end{array}
$$

$(\forall\alpha.\alpha\rightarrow\alpha)\rightarrow\alpha$ e $(\forall\alpha.\alpha\rightarrow\delta)\rightarrow\delta$ não têm um generalizador comum. Logo, temos termos com tipos diferentes que não podemos generalizar para um tipo comum, o que mostra a não existência de tipos principais.

3.3 Discussão

Neste capítulo, apresentámos o sistema de Damas-Milner como modelo do polimorfismo paramétrico. Este estudo prévio do polimorfismo é necessário para, a partir daqui, focar dois assuntos fundamentais:

- O estudo de modelos alternativos de polimorfismo como os sistemas com intersecções;
- Um estudo mais geral das propriedades que o próprio conceito de polimorfismo impõe aos sistemas de tipos e até que ponto os modelos existentes respeitam essas propriedades.

O primeiro ponto vai ser apresentado no próximo capítulo. O segundo ponto surge como consequência de uma metodologia mais geral que permite analisar vários sistemas de tipos sob o ponto de vista semântico. A esta metodologia chamamos sistemas com restrições e vai ser apresentada nos capítulo 5 e 6.

Capítulo 4

Sistemas de Tipos com Intersecções

4.1 Introdução

Este capítulo foca sistemas de tipos com intersecções, também conhecidos como sistemas de Turino ([CD80] e [CDV81]).

Tendo em conta que o objectivo último deste trabalho é a caracterização semântica de sistemas de tipos, o estudo de tipos com intersecções foi escolhido, fundamentalmente pelas seguintes razões:

- São sistemas em que a inferência de tipos não é um problema decidível, sendo, portanto, um laboratório para o estudo de caracterizações semânticas de sistemas de tipos e de propriedades como a da existência de tipos principais em sistemas em que não podemos obter algoritmos para os construir;
- A noção de instância, nestes sistemas, toma um carácter mais geral. Aqui a instância não é definida somente à custa da substituição, mas sim da composição da substituição com outras operações, de entre as quais assume particular importância a expansão. Neste sistema se π for o tipo principal de uma expressão e, os tipos τ tais que existe uma base B tal que $B \vdash e : \tau$ são da forma $\tau \in O(\pi)$, onde O é uma operação diferente da substituição. Isto permite um estudo aprofundado de casos em que não temos integridade do tipo principal, porque a própria noção de instância não é íntegra, isto é, $\vdash e : \pi$ não implica $\vdash e : O(\pi)$.

O estudo de sistemas de tipos com intersecções, tendo em conta estes dois últimos factores, permitiu-nos:

- Abordar o problema da existência de tipos principais de um modo semântico, partindo do princípio que se trata de um problema não decidível.
- Clarificar a necessidade de alterações ao sistema de Coppo-Dezani, justificadas pela necessidade de garantir a integridade da noção de instância e, consequentemente, do tipo principal.
- Salientar e justificar a necessidade da propriedade da integridade da noção de instância em sistemas de tipos.

Este capítulo resume-se à apresentação das versões mais usadas de sistemas de tipos com intersecções e da necessidade de cada versão como resolução de um problema existente na versão anterior.

4.2 Intersecção como Polimorfismo

No polimorfismo paramétrico, um esquema de tipos representa um conjunto de tipos que podem ser atribuidos a um termo. $\forall\alpha.\alpha \rightarrow \beta$ representa todos os tipos que podemos obter substituindo α por qualquer tipo. É fácil verificar que um tipo polimórfico representa um número infinito de tipos, pois a variável genérica α pode assumir qualquer valor.

Vamos supor que limitamos o domínio da substituição aplicada a α indicando explicitamente qual o conjunto de tipos pela qual pode ser substituida. Teriamos algo como $\forall_{\alpha\in\{\sigma_1,\ldots,\sigma_n\}}.\alpha \rightarrow \beta$ ou mais simplesmente $\{\sigma_1,\ldots,\sigma_n\} \rightarrow \beta$. Este é o tipo de uma função que retorna um objecto de tipo β quando aplicada a um objecto que tem simultaneamente tipos $\{\sigma_1,\ldots,\sigma_n\}$.

A semântica de um esquema de tipos $\|\forall_\alpha.\sigma\|$ é a intersecção das semânticas para todas as instâncias da variável α. Também aqui a semântica de $\{\sigma_1,\ldots,\sigma_n\}$, $\|\sigma_1,\ldots,\sigma_n\| = \bigcap_i \|\sigma_i\|$. Daí o nome de tipos com intersecções dado a estes sistemas e chamar intersecções (representadas por $\sigma_1 \cap \ldots \cap \sigma_n$) ao que nós começámos por chamar conjuntos de tipos. Originalmente, eram denominados sequências de tipos.

Salientamos o facto de que sistemas com intersecção não são mais do que uma forma de polimorfismo em que é afirmado explicitamente que um dado termo tem simultaneamente tipos $\sigma_1,\ldots,\sigma_n$ (representado por $\sigma_1 \cap \ldots \cap \sigma_n$).

4.3 O Sistema de Coppo-Dezani

Em 1980 foi apresentado o primeiro sistema de tipos com intersecções. ([CD80]).

4.3.1 Motivação

No sistema de Curry há termos do λ-calculus correctos sob o ponto de vista semântico que não admitem tipo. Um exemplo é $\lambda x.xx$. Para este termo admitir tipo x tem que ter simultaneamente tipos $\alpha \to \beta$ e α, o que faz com que x tenha que ser usado polimorficamente na regra de abstracção ($\to$-INTRO). O sistema de Damas e Milner também não resolve este problema, pois não permite tipos polimórficos na abstracção.

É fácil verificar que expressões que não admitem tipo podem ser aplicadas e o resultado dessa aplicação pode ser tipificável. Por exemplo, se aplicarmos $\lambda x.xx$ a $\lambda y.y$ temos $(\lambda x.xx)(\lambda y.y)$ que tem tipo $\alpha \to \alpha$.

No sistema de Coppo-Dezani, este problema é resolvido permitindo mais do que um tipo para variáveis numa abstracção e, consequentemente, mais do que um tipo para o argumento de uma função na regra da aplicação ($\to$ ELIM).

Neste sistema os conjuntos de intersecções e de tipos são definidos por:

1. Todas as variáveis de tipos são tipos;

2. Se $\sigma_1, \ldots, \sigma_n$ $(n \leq 1)$ são tipos $\bar{\sigma} \equiv \sigma_1 \cap \ldots \cap \sigma_n$ é uma intersecção (originalmente chamada sequência);

3. Se $\sigma_1 \cap \ldots \cap \sigma_n$ é uma intersecção e σ um tipo, então $\sigma_1 \cap \ldots \cap \sigma_n \to \sigma$ é um tipo.

Repare-se que o sistema original é construido de acordo com a motivação inicial que é permitir mais do que um tipo como argumento de uma função. Isto faz com que só sejam necessárias intersecções no lado esquerdo de $\to$.

As regras de inferência de tipos são:

$$AXIOM \quad A \cup \vdash x : \sigma_i \text{ ,se } \sigma_i \in \bar{\sigma} \text{ e } x : \bar{\sigma} \in A$$

$$\to ELIM \quad \frac{A \vdash M : \sigma_1 \cap \ldots \cap \sigma_n \to \sigma, A \vdash N : \sigma_1 \ldots A \vdash N : \sigma_n}{A \vdash MN : \sigma}$$

$$\to INTRO \quad \frac{A \cup \{x : \sigma_1 \cap \ldots \cap \sigma_n\} \vdash M : \sigma}{A \vdash \lambda x.M : \sigma_1 \cap \ldots \cap \sigma_n \to \sigma} \quad (a)$$

(a) Neste caso $x : \sigma_1 \cap \ldots \cap \sigma_n$ é a única declaração para x da qual depende $M : \sigma$.

4.3.2 O Sistema de Coppo-Dezani para o λI-calculus

O λI-calculus, desenvolvido por Church ([Chu41]), é uma restrição ao λ-calculus em que, para todos os termos da forma $\lambda x.M$, x ocorre livre em M.

Definição 4.1 *O conjunto de λI-termos, Λ_I, é definido indutivamente por:*

1. $x \in \Lambda_I$
2. $M \in \Lambda_I$ *e* $x \in FV(M) \Rightarrow \lambda x.M \in \Lambda_I$
3. M *e* $N \in \Lambda_I \Rightarrow MN \in \Lambda_I$

Neste sistema, demonstrou-se em [CD80] que para o λI-calculus o sistema é fechado por igualdade β, isto é, $B \vdash M : \sigma \wedge M \stackrel{\beta}{=} N \Rightarrow B \vdash N : \sigma$.

No artigo acima referido, demonstrou-se também que todos os termos em forma normal são tipificáveis e que, se um termo admite um tipo, então, admite forma normal. A partir destas três propriedades, conclui-se facilmente que um termo do λI-calculus é tipificável se e só se admite forma normal. Como corolário deste resultado, temos a indecidibilidade do sistema (dado que saber se um termo do λI-calculus admite forma normal é um problema indecidível).

Note-se que o sistema de Curry não é fechado por igualdade β no λI-calculus:

Exemplo 4.1 $(\lambda x.xx)(\lambda y.y) \stackrel{\beta}{=} \lambda y.y$, $\not\vdash (\lambda x.xx)(\lambda y.y) \wedge \vdash \lambda y.y : \alpha \to \alpha$.

Neste exemplo $(\lambda x.xx)(\lambda y.y)$ não é tipificável no sistema de Curry, porque $\lambda x.xx$ não admite tipo. Já no sistema de Coppo-Dezani $\vdash \lambda x.xx : (\alpha \cap \alpha \to \beta) \to \beta$, pelo que $\vdash (\lambda x.xx)(\lambda y.y) : \alpha \to \alpha$.

No λ-calculus, tanto o sistema de Curry como o sistema de Coppo-Dezanni não são fechados por igualdade β. Podemos ter $B \vdash M[N/x] : \sigma$ e $B \not\vdash (\lambda x.M)N : \sigma$. Isto porque no λ-calculus a variável x pode não ocorrer livre em M e, neste caso, N não é um subtermo de $M[N/x]$. Se N não for tipificável, podemos ter um tipo para $M[N/x]$ e não ter tipo para $(\lambda x.M)N$.

Exemplo 4.2 *$(\lambda x \lambda y.y)((\lambda x.xx)(\lambda x.xx)) \stackrel{\beta}{=} \lambda y.y$, $\vdash \lambda y.y : \alpha \to \alpha$ e o primeiro termo não é tipificável nem no sistema de Curry nem no sistema de Coppo-Dezani, porque $(\lambda x.xx)(\lambda x.xx)$ não é tipificável nestes sistemas.*

4.4 Outros Sistemas de Tipos com Intersecções

4.4.1 O Sistema de Coppo-Dezani-Venneri

Em [CDV81] foi apresentado um sistema de tipos para o λ-calculus fechado por igualdade β.

Como vimos na secção anterior, esta propriedade não se verificava, pois, ao expandir $M[N/x]$ para $(\lambda x.M)N$, x pode não ocorrer em M e N pode não ser tipificável, o que faz com que apesar de $M[N/x]$ (neste caso igual a M) possa admitir um tipo, o mesmo não aconteça para $(\lambda x.M)N$. Neste sistema, o problema é resolvido através da definição de um tipo universal ω que tipifica todos os termos do λ-calculus. Neste caso, quando $B \vdash M : \sigma$ e x não ocorre livre em M temos $B \vdash \lambda x.M : \omega \to \sigma$, e por definição $B \vdash N : \omega$ para todo o N. Logo $B \vdash (\lambda x.M)N : \sigma$. O sistema é obviamente indecidível, pois atribuir um tipo a um termo é um problema equivalente ao de saber se o termo tem forma normal.

O sistema de Coppo-Dezani-Venneri é constituido pelas seguintes regras de inferência:

$AXIOM \quad A \vdash x : \sigma$,se $\sigma \in A$

$$\omega \qquad \frac{}{A \vdash M : \omega}$$

$$\rightarrow ELIM \qquad \frac{A \vdash M : \sigma_1 \cap \ldots \cap \sigma_n \rightarrow \sigma, A \vdash N : \sigma_1 \cap \ldots \cap \sigma_n}{A \vdash MN : \sigma}$$

$$\rightarrow INTRO \qquad \frac{A \cup \{x : \sigma_1, \ldots, x : \sigma_n\} \vdash M : \sigma}{A \vdash \lambda x.M : \bar{\tau} \rightarrow \sigma} \quad (a)$$

$$\bigcap INTRO \qquad \frac{A \vdash M : \sigma_1, \ldots, A \vdash M : \sigma_n}{A \vdash M : \sigma_1 \cap \ldots \cap \sigma_n}$$

(a) Neste caso, $x : \sigma_1, \ldots, x : \sigma_n$ são as únicas declarações para x das quais depende $e : \sigma$ e $\bar{\tau}$ é uma intersecção que contém $\{\sigma_1, \ldots, \sigma_n\}$.

4.4.2 O Sistema de Barendregt-Coppo-Dezani

Mais versões de sistemas de tipos com intersecções apareceram desde então. Uma das mais importantes é o chamado sistema de Barendregt-Coppo-Dezani, [BCD83]. As inovações introduzidas por este sistema foram:

1. Possibilidade de ter intersecções no lado direito de $\rightarrow$;
2. Definição de uma relação de ordem entre tipos, e baseada nesta relação, introdução de uma nova regra de inferência:

$$(\leq) \quad \frac{A \vdash M : \sigma, \sigma \leq \tau}{A \vdash M : \tau}$$

Este sistema tem as mesmas propriedades do anterior. Uma propriedade torna-o particularmente interessante: o conjunto de tipos deriváveis para um dado termo é fechado pela intersecção e fechado à direita para $\leq$. Esse conjunto fornece uma semântica completa para o λ-calculus, isto é, se a semântica de um termo for o conjunto de tipos inferíveis para esse termo então $\vdash M : \tau \Leftrightarrow \|M\| \in \|\tau\|$.

4.4.3 Sistema de Tipos Estritos

Este sistema apresentado em [Bak92] é uma restrição ao sistema de Barendregt-Coppo-Dezani, e em relação a este, difere nos seguintes aspectos:

1. A relação $\leq$ e a regra de inferência $\leq$ não existem;
2. O conjunto de tipos volta a ser o apresentado no trabalho original do Coppo-Dezani. Deixa de ser permitido ter intersecções no lado direito de $\rightarrow$.

O sistema de tipos estrito é definido pelas seguintes regras de inferência:

$$AXIOM \qquad A \vdash x : \sigma_i \text{ , se } \sigma_i \in \bar{\sigma} \text{ e } x : \bar{\sigma} \in A$$

$$\rightarrow ELIM \qquad \frac{A \vdash M : \sigma_1 \cap \ldots \cap \sigma_n \rightarrow \sigma, A \vdash N : \sigma_1 \ldots A \vdash N : \sigma_n}{A \vdash MN : \sigma} \quad (n \geq 0)$$

$$\rightarrow INTRO \quad \frac{A \cup \{x : \sigma_1 \cap \ldots \cap \sigma_n\} \vdash M : \sigma}{A \vdash \lambda x.M : \sigma_1 \cap \ldots \cap \sigma_n \rightarrow \sigma} \quad (a)$$

(a) Neste caso $x : \sigma_1 \cap \ldots \cap \sigma_n$ é a única declaração para x da qual depende $M : \sigma$.

Neste sistema, o tipo universal ω é representado por uma intersecção vazia. Também não temos regra de $\bigcap INTRO$. O que existe no sistema estrito é uma segunda inferência $\vdash_S$ que é definida por $B \vdash_S M : \sigma_1 \cap \ldots \cap \sigma_n$ se e só se $B \vdash M : \sigma_i$, $1 \leq i \leq n$. Neste sistema, mantém-se a propriedade de fecho por igualdade β.

4.5 Tipos Principais

Um tipo principal é uma representação sintáctica de todos os tipos inferíveis para um determinado termo. Se, para qualquer termo tipificável M, existe uma base P tal que $P \vdash M : \pi$ e π é o tipo principal de M, o sistema de tipos tem a propriedade do tipo principal.

Originalmente no trabalho de Hindley, [Hin69], qualquer tipo pode ser obtido a partir do tipo principal por substituição. Se π for o tipo principal de M e τ for qualquer tipo tal que existe uma base B tal que $B \vdash M : \tau$, então, existe uma substituição S tal que $\tau = S(\pi)$. Isto acontece, porque a estrutura da derivação de tipos é determinada somente pela estrutura sintáctica dos termos. Neste caso, as árvores de derivação de tipos de um determinado termo são isomorfas. Só variam os nós, mas a estrutura de todas as árvores mantém-se.

O mesmo não acontece em sistemas de tipos com intersecções. No caso particular do sistema de Coppo-Dezani, isto não acontece por duas razões:

1. Na aplicação de uma regra de aplicação, podemos substituir o tipo do argumento de uma função por uma intersecção de tipos, desde que todos sejam tipos do argumento.

   Exemplo 4.3 *Vamos supor que uma árvore de derivação termina numa regra de $\rightarrow ELIM$:*

$$\begin{tikzcd}
B \arrow[d] & & B \arrow[d] \\
\vdash M : \sigma \rightarrow \tau \arrow[dr] & & \vdash N : \sigma \arrow[dl] \\
& \vdash MN : \tau &
\end{tikzcd}$$

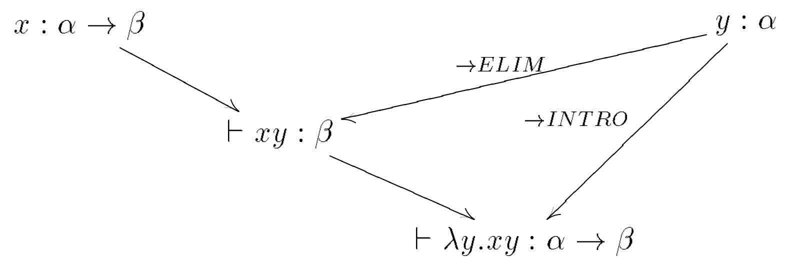
   e que existe uma outra derivação:

$$\begin{tikzcd}
A \arrow[d] & & A \arrow[d] & A \arrow[d] \\
\vdash M : \sigma_1 \cap \sigma_2 \rightarrow \tau \arrow[dr] & & \vdash N : \sigma_1 \arrow[dl] & \vdash N : \sigma_2 \arrow[dll] \\
& \vdash MN : \tau & &
\end{tikzcd}$$

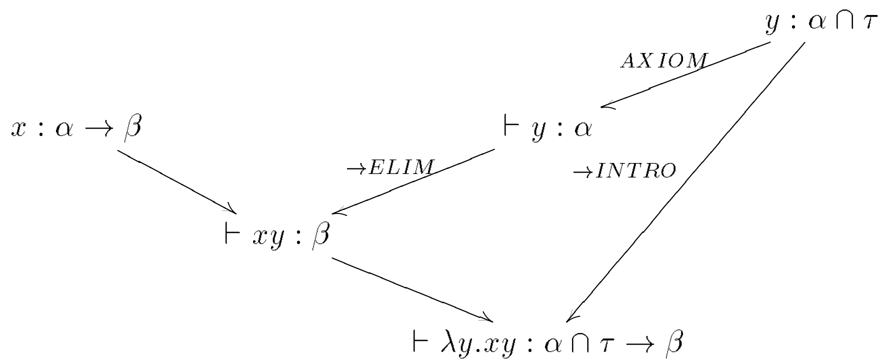
   Estas duas derivações para o mesmo tipo de MN não são isomorfas. Isto porque, na regra de aplicação, podemos mais do que um tipo para N.

2. Podemos aumentar o comprimento de uma árvore de derivação que termina numa regra de abstracção, acrescentando tipos na base por intersecção e aplicando a regra do axioma.

Exemplo 4.4 *Consideremos as seguintes árvores de derivação:*



e



Existem bases B_1 e B_2 tais que $B_1 \vdash \lambda y.xy : \alpha \to \beta$ e $B_2 \vdash \lambda y.xy : \alpha \cap \tau \to \beta$ e as duas árvores não são isomorfas.

Estes dois exemplos correspondem a operações nos tipos diferentes da substituição. No entanto, o problema do tipo principal é generalizável para sistemas de tipos com intersecção como:

Dado um termo M existe algum tipo π tal que $\exists_B . B \vdash M : \tau \Rightarrow \tau = O(\pi)$, para uma operação O? (não necessariamente a substituição).

Em [CDV80] é definida uma operação, a expansão, que juntamente com a substituição permite chegar a qualquer tipo inferível a partir do tipo principal.

Esta operação consiste na substituição de tipos por um certo numero de cópias. Isto faz com que a definição de expansão seja muito complexa. Expansões de tipos correspondem à duplicação de derivações apresentada no exemplo 4.3.

Esta operação tal como é definida [CDV80] não é íntegra, isto é, para uma dada base B, podemos ter $B \vdash M : \tau$ e τ' obtido a partir de τ por expansão e não ser possível inferir τ' para M. Posteriormente, este problema foi resolvido em [RV84] e [Bak93b] desdobrando a operação em expansão e elevação. A primeira corresponde à duplicação de subderivações, ilustrada no exemplo 4.3 e a segunda à aplicação da regra $\leq$, a que corresponde o exemplo 4.3. As definições que se seguem são baseadas em [Bak93b].

4.5.1 Expansão

A expansão é uma operação que afecta todo o contexto de uma derivação. A expansão de um subtipo μ de σ numa derivação de σ para um termo M a partir de uma base B afecta σ e B. Como já foi dito, a expansão consiste na substituição de um tipo μ por n cópias desse tipo. Portanto, para determinar uma expansão, precisamos do tipo a expandir μ, do numero de cópias a efectuar n, do tipo σ do qual μ é subtipo e da base B que origina a derivação de σ. Sendo assim, a expansão é definida por uma função $< \mu, n, B, \sigma >: T \rightarrow T$, onde T é o conjunto de tipos.

Repare-se que garantir que o resultado de uma expansão ainda seja um tipo torna a definição da operação bastante complexa quando aplicada ao conjunto de tipos originalmente apresentados por Coppo e Dezani. Por exemplo, suponhamos que τ é um subtipo de σ que vai ser substituido pela intersecção de n cópias. Seja $\tau' \rightarrow \tau$ um subtipo de σ. Não podemos substituir τ por $\tau_1 \cap \ldots \cap \tau_n$ pois, nesse caso, o resultado daria origem a $\tau' \rightarrow \tau_1 \cap \ldots \cap \tau_n$ que não é um tipo válido. Neste caso, a solução é expandir $\tau' \rightarrow \tau$ na totalidade dando origem a $(\tau_1' \rightarrow \tau_1) \cap \ldots \cap (\tau_n' \rightarrow \tau_n)$. O problema reside na expansão de variáveis que ocorrem no lado direito de $\rightarrow$. Estas variáveis são definidas como *últimas variáveis*:

Definição 4.2 *A última variável de um tipo é definida como:*

- *A última variável de uma variável α é α.*
- *A última variável de $\tau_1 \cap \ldots \cap \tau_n \rightarrow \tau$ é a última variável de τ.*

O conjunto $V_\mu(< B, \sigma >)$, de todas as variáveis que podem ser afectadas pela expansão, é definido por:

1. Se α ocorre em μ, então $\alpha \in V_\mu(< B, \sigma >)$;
2. Seja τ um subtipo de σ ou de qualquer tipo predicado em B. Seja α a última variável de τ. Se $\alpha \in V_\mu(< B, \sigma >)$, então, para todas as variáveis β que ocorrem em τ, $\beta \in V_\mu(< B, \sigma >)$.

Estamos agora em condições de definir a operação de expansão:

Definição 4.3 *Suponhamos que* $V_\mu(< B, \sigma >) = \{\alpha_1, \ldots, \alpha_n\}$. *Sejam* $\alpha_1^1, \ldots, \alpha_1^n, \ldots, \alpha_m^1, \ldots \alpha_m^n$ $m \times n$ *variáveis novas. Seja* S_i *a substituição que substitui* α_j *por* α_j^i.

1. $< \mu, n, B, \sigma > (\alpha) = S_1(\alpha) \cap \ldots \cap S_n(\alpha)$, *se a última variável de* α *pertence a* $V_\mu(< B, \sigma >)$;

2. $< \mu, n, B, \sigma > (\alpha \to \beta) =< \mu, n, B, \sigma > (\alpha) \to< \mu, n, B, \sigma > (\beta)$, *se a última variável de* β *não pertence a* $V_\mu(< B, \sigma >)$;

3. $< \mu, n, B, \sigma > (\alpha) = \alpha$, *se* $\alpha \notin V_\mu(< B, \sigma >)$;

4. $< \mu, n, B, \sigma > (\tau_1 \cap \ldots \cap \tau_n) =< \mu, n, B, \sigma > (\tau_1) \cap \ldots \cap < \mu, n, B, \sigma > (\tau_n)$.

Esta definição é generalizável para Bases e pares $< base, tipo >$ por:

1. $< \mu, n, B, \sigma > (B) = \{x :< \mu, n, B, \sigma > (\rho) | x : \rho \in B\}$;

2. $< \mu, n, B, \sigma > (< B, \sigma >) = (<< \mu, n, B, \sigma > (B), < \mu, n, B, \sigma > (\sigma) >)$.

4.5.2 Elevação

Em [BCD83] foi definida uma regra $\leq$ que torna inferíveis todos os tipos τ tais que $\sigma \leq \tau$ se σ for inferível para um determinado termo. Essa regra $\leq$ é similar à regra de instância do sistema de Damas e Milner. Como já foi referido, uma intersecção $\tau_1 \cap \ldots \cap \tau_n$ pode ser vista como polimorfismo no sentido em que $M : \tau_1 \cap \ldots \cap \tau_n$ significa que M tem simultaneamente tipos $\tau_1, \ldots \tau_n$. No polimorfismo paramétrico, existe a regra de instância:

$$\frac{A \vdash M : \forall_\alpha.\sigma}{A \vdash M : \sigma[\alpha/\tau]}$$

que garante explicitamente que qualquer instância de um tipo inferível é ainda inferível para um determinado termo. No contexto de sistemas com intersecção, é legítimo afirmar que o equivalente à noção de instância é dizer que qualquer σ_i, $1 \leq i \leq n$ é uma instância de $\sigma_1 \cap \ldots \cap \sigma_n$. A partir desta ideia é definida a relação $\leq$ que tem como significado $\sigma_1 \leq \sigma_2$ se σ_2 for uma instância (no contexto de tipos com intersecção) de σ_1. A regra $\leq$ toma a forma:

$$(\leq) \quad \frac{A \vdash M : \sigma, \sigma \leq \tau}{A \vdash M : \tau}$$

Várias definições da relação $\leq$ foram apresentadas, todas com o significado "instância de". A que vamos apresentar aqui é definida em [Bak93b]:

Definição 4.4 *Define-se a relação $\leq$ no $T \bigcup I$, onde T é o conjunto de tipos e $I = \{\tau_1 \cap ... \cap \tau_n | \forall_i . \tau_i \in T\}$, do seguinte modo:*

1. $\sigma_1 \cap \ldots \cap \sigma_n \leq \sigma_i,\ 1 \leq i \leq n$;
2. $1 \leq i \leq n, \sigma \leq \sigma_i \Rightarrow \sigma \leq \sigma_1 \cap \ldots \cap \sigma_n$;
3. $\sigma \leq \tau \leq \rho \Rightarrow \sigma \leq \rho$;
4. $\rho \leq \sigma \wedge \tau \leq \mu \Rightarrow \sigma \rightarrow \tau \leq \rho \rightarrow \mu$.

Note-se que em vários sistemas a regra $\leq$ surge sob outras formas, por exemplo, na regra do $AXIOMA$:

$$AXIOM \quad A \vdash x : \sigma_i \text{ ,se } \sigma_i \in \bar{\sigma} \text{ e } x : \bar{\sigma} \in A$$

estamos a utilizar o ponto 1. da definição de $\leq$ aplicado a variáveis do λ-calculus.

A relação $\leq$ é facilmente extensível a bases:

$$B_1 \leq B_2 \Leftrightarrow \forall_{x:\sigma_2 \in B_2} \exists_{x:\sigma_1 \in B_1} . \sigma_1 \leq \sigma_2$$

Finalmente, definimos a operação de elevação:

Definição 4.5 *Dados dois pares $< B_1, \tau_1 >$ e $< B_2, \tau_2 >$, dizemos que $< B_2, \tau_2 >$ é um elevação de $< B_1, \tau_1 >$ se e só se:*

- $B_2 \leq B_1$;
- $\tau_1 \leq \tau_2$.

Por exemplo, $\alpha \cap \beta \to \tau$ é um elevação de $\alpha \to \tau$.

Como foi demonstrado em [RV84], uma composição finita de substituições, expansões e elevações define uma operação, a que se chama cadeia, que é usada para chegar a todos os tipos inferíveis para um termo a partir do tipo principal, quando este existe.

4.5.3 Existência de Tipos Principais

A existência de tipos principais foi demonstrada para vários sistemas de tipos com intersecção com o tipo mais geral ω. ([CDV80], [RV84] e [Bak93b]).

A técnica usada na demonstração da existência de tipos principais é, em todos os casos, a seguinte:

1. Define-se o tipo principal para termos em forma normal;
2. Prova-se que qualquer tipo inferido para um termo em forma normal é obtido a partir do tipo principal por uma composição de substituições, expansões e elevações;
3. Usa-se o facto destes sistemas de tipos serem fechados por igualdade β para generalizar o resultado para termos arbitrários.

Em sistemas de tipos sem a constante ω, não podemos usar este método, pois a propriedade do fecho por igualdade β não se verifica. Isto faz com que o estudo de tipos principais para estes sistemas quase não exista. É referida a existência de tipos principais para um sistema sem ω e com a regra de $\leq$ usando um algoritmo de unificação em [Ron88]. Demonstrámos que basta ter uma regra de $\leq$ aplicada a variáveis do λ-calculus para obter o mesmo resultado.

4.5.4 Tipos Principais e Unificação

O algoritmo de unificação de Robinson é a base de vários algoritmos de inferência de tipos. ([Hin69] e [DM82]).

Para sistemas com intersecções existe um procedimento semi-decidível, baseado no algoritmo da unificação ([Ron88]), que, dado um termo, retorna o seu tipo principal se e só se o termo for fortemente normalizável:

Teorema 4.1 (Ronchi della Rocca) *Seja PP o procedimento de inferência de tipos e M um termo do λ-calculus. Então $PP(M) =< P, \pi > \Leftrightarrow M$ é fortemente normalizável e $< P, \pi >$ é o tipo principal de M.*

A inferência é baseada no algoritmo de unificação em que a noção de susbstituição é generalizada para composições de expansões, substituições e elevações.

As duas propriedades seguintes são necessárias à demonstração da existência de tipos principais em sistemas sem a constante ω.

Proposição 4.1 (Ronchi della Rocca) *Existe um procedimento, UNIFY, que dados dois tipos τ e τ', sucede com uma cadeia c que unifica τ e τ', isto é $c(\tau) = c(\tau')$, se existir uma cadeia que unifique τ e τ' ou não termina caso contrário.*

Antes do teorema que garante a completitude do procedimento UNIFY definimos a noção de fecho de dois pares $< base, tipo >$:

Definição 4.6 *Sejam $< B_1, \pi_1 >$ e $< B_2, \pi_2 >$ dois pares disjuntos. O fecho de $< B_1, \pi_1 >$ e $< B_2, \pi_2 >$ é um par $< \sigma, \tau >$ tal que:*

- $\sigma \equiv \sigma_1 \to \ldots \to \sigma_n \to \pi_1$;
- $\tau \equiv \tau_1 \to \ldots \to \tau_n \to \pi_2 \to \alpha$.

sendo α uma variável nova e se $x_1, \ldots, x_n$ forem os sujeitos da declaração em B_1 e B_2 e se α_i e β_i forem os predicados de x_i em B_1 e B_2 respectivamente, se existirem ou uma variável nova se não existirem, então $\sigma_i \equiv \alpha_i \wedge \phi_i$ e $\tau_i \equiv \eta_i \wedge \beta_i$, onde ϕ_i e η_i são variáveis novas.

Teorema 4.2 (Ronchi della Rocca, Completitude da unificação) *Seja $< \sigma, \tau >$ o fecho de dois tipos principais distintos. Se existe uma cadeia c tal que $c(\sigma) = c(\tau)$, então $UNIFY(\tau, \sigma) = c'$ e $c = c' \circ c^*$ para algum c^*, isto é, c' é o unificador mais geral de τ e σ.*

4.6 Discussão

O sistema original de Coppo-Dezani, [CD80], foi apresentado como uma extensão e simplificação do polimorfismo paramétrico de Damas e Milner. É apresentado como uma extensão a esse sistema, na medida em que tipifica mais termos do que o sistema anterior. Por exemplo, o termo $\lambda x.xx$, que não é tipificável em ML tem tipo $(\alpha \cap (\alpha \rightarrow \beta) \rightarrow \beta)$ no sistema de tipos com intersecções.

É apresentado como uma simplificação pois evita o uso de esquemas de tipos e, consequentemente, das regras de generalização e de instância. Apesar do facto de se ter provado ser um sistema indecidível, o que faz com que o seu uso no desenho de linguagens de programação não seja viável, poderíamos ser levados a pensar que restrições decidíveis do sistema inicial poderiam ser viáveis no desenho de sistemas de tipos com aplicação na área das linguagens de programação. Para isso, contribui a ideia generalizada da existência de tipos principais para o sistema original de tipos com intersecções. De facto, uma extensão ao sistema inicial com a propriedade do fecho por igualdade β, isto é, $A \vdash M_1 : \tau_1$ e $M_1 \stackrel{\beta}{=} M_2$ então $A \vdash M_2 : \tau_2$, foi introduzida em [CDV80] com a propriedade do tipo principal. Os autores do sistema original referem no artigo [Cop80] que com uma adaptação simples da demonstração usada em [CDV80] podemos concluir que o sistema original tem a propriedade do tipo principal. Tal não se verifica e, mesmo que se verificasse, há propriedades do sistema original de Coppo-Dezani que fazem com que os tipos principais não possam ser utilizáveis na prática. Acontece que a demonstração da existência de tipos principais usada em [CDV80] assenta na seguinte ideia:

1. Define-se o tipo principal para termos em forma normal;
2. Prova-se que qualquer tipo inferido para um termo em forma normal é obtido a partir do tipo principal por uma composição de substituições, expansões e elevações;
3. Usa-se o facto destes sistemas de tipos serem fechados por igualdade β para generalizar o resultado para termos arbitrários.

Ora, esta demonstração não pode ser aplicada ao sistema descrito em [Cop80] pois ele não é fechado por igualdade β. Mais do que isso, este sistema não é íntegro com relação à noção de instância. Isto quer dizer que podemos ter $A \vdash M : \sigma$ e $A \not\vdash M : \sigma'$ para um tipo σ' cuja semântica está incluida na semântica de σ, isto é, σ' é uma instância (sob o ponto de vista semântico) de σ. Uma análise pormenorizada destes problemas vai ser apresentada no capítulo 7.

Capítulo 5

Sistemas de Tipos com Restrições

5.1 Introdução

Neste capítulo, apresentamos um novo sistema de tipos assente directamente em restrições induzidas pela relação semântica entre termos e tipos. Como motivação fundamental deste trabalho, surge um estudo das razões da falta de completitude dos sistemas de tipos estudados previamente. Esse estudo levou-nos a identificar razões muito diferentes. Atente-se nos seguintes exemplos:

1. Considere-se o termo $M \equiv \lambda x.xx$. Para tipificar M nos sistemas de Curry, o tipo de x teria que ser simultaneamente da forma α e $\alpha \to \beta$. Como α e $\alpha \to \beta$ não são unificáveis num único tipo, o sistema de Curry não consegue tipificar $\lambda x.xx$. Repare-se que, sob o ponto de vista semântico, não é possivel exprimir o tipo para este termo usando a linguagem de Curry. Qualquer tipo τ, tal que $\models \lambda x.xx : \tau$ tem que ser tal que

   $$\forall_\varepsilon \forall_\rho . E\|\lambda x.xx\|\rho \in T\|\tau\|\varepsilon$$

   Isto implica que:

   $$\forall_\varepsilon \exists_{\alpha,\beta}[T\|\tau\|\varepsilon \subseteq T\|\alpha \to \beta\|\varepsilon \wedge T\|\tau\|\varepsilon \subseteq T\|\alpha\|\varepsilon]$$

   Para esta última fórmula se verificar para todo o ε, τ tem que ser simultaneamente menor ou igual a um tipo $\alpha \to \beta$ e a α e caímos no mesmo motivo que leva à não inferência de tipo para $\lambda x.xx$. No entanto, esta impossibilidade leva a problemas mais graves. Considere-se o termo $(\lambda x.xx)(\lambda y.y)$. Temos

$\models (\lambda x.xx)(\lambda y.y) : \alpha \to \alpha$, pois $(\lambda x.xx)(\lambda y.y)$ reduz-se a $\lambda y.y$ e como é impossível tipificar $\lambda x.xx$ não podemos tipificar $(\lambda x.xx)(\lambda y.y)$. Aqui temos tipos válidos sob o ponto de vista semântico que o sistema de Curry não consegue tipificar. Como vimos no capítulo 4, há sistemas que resolvem este problema à custa da perda da decidibilidade.

2. Considere-se agora o termo $\lambda x.x$. De um modo geral $\not\models \lambda x.x : \alpha \to \beta$ pois podemos ter α e β tais que $\forall_\varepsilon[\varepsilon(\beta) \subset \varepsilon(\alpha)]$ e nesse caso $E\|\lambda x.x\| \not\subseteq T\|\alpha \to \beta\|\varepsilon$. No entanto, em todos os casos tais que $\forall_\varepsilon[\varepsilon(\alpha) \subseteq \varepsilon(\beta)]$ temos $E\|\lambda x.x\| \subseteq T\|\alpha \to \beta\|\varepsilon$ e o sistema de Curry não é suficientemente geral para capturar esses casos. Concretizando, se $\forall_\varepsilon[\varepsilon(\alpha) \subseteq \varepsilon(\beta)]$, é semanticamente válido dizer que a identidade transforma elementos de $\varepsilon(\alpha)$ em elementos de $\varepsilon(\beta)$, logo, nesse caso, teríamos $\alpha \to \beta$ como tipo de $\lambda x.x$, o que não acontece no sistema de Curry.

Repare-se que, no primeiro exemplo, há duas restrições semânticas ao tipo τ de $\lambda x.xx$, que são:

1. $T\|\tau\|\varepsilon \subseteq T\|\alpha \to \beta\|\varepsilon$
2. $T\|\tau\|\varepsilon \subseteq T\|\alpha\|\varepsilon$

No segundo caso, temos uma restrição semântica ao tipo $\alpha \to \beta$ para $\lambda x.x$ que é $T\|\alpha\|\varepsilon \subseteq T\|\beta\|\varepsilon$. Isto levou-nos à definição de um novo sistema de tipos que tivesse em conta essas restrições. Demonstrámos que a resolução das restrições tomando as soluções dadas por igualdade nos leva exactamente ao sistema de Curry, isto é, o sistema de Curry não é mais do que um modo de resolver um conjunto de restrições impostas pela relação semântica entre termos e tipos. Esse modo consiste em admitir somente soluções que satisfaçam igualdades em todas as inequações do sistema, ou seja, só admite soluções que podem ser obtidas usando o algoritmo da unificação de Robinson. [Rob65]. Caso estas soluções não existam, o sistema não tipifica o termo em questão. Nos capítulos 6 e 7, fazemos um estudo similar em relação a outros sistemas de tipos, o que vai permitir clarificar, sob o ponto de vista semântico, noções tão importantes como o polimorfismo e saber o porquê de alguns problemas de extensões ao sistema de Curry.

Neste capítulo, começamos por apresentar o que se entende por restrições semânticas ao conjunto de tipos admissíveis para um determinado termo. Em seguida, apresentamos o novo sistema de tipos e a demonstração da sua integridade com respeito à semântica apresentada no segundo capítulo. Mostramos, finalmente, a relação entre o novo sistema e o sistema de Curry.

5.2 Tipos Semânticos

Partindo de uma visão puramente semântica da relação entre termos e tipos, temos, de um lado, um conjunto de valores V, cada um deles com determinadas propriedades. Entende-se formalmente como propriedade (ou tipo) aquilo que é comum a um conjunto de valores com determinadas características, os ideais. Isto é, um ideal define uma propriedade comum a todos os elementos desse ideal. Partindo de uma linguagem de tipos L, só algumas propriedades são exprimíveis em L. Ao definir a relação entre tipos e termos limitamos o subconjunto de propriedades que conseguimos exprimir em L aquelas que conseguimos associar aos respectivos termos. Vamos ver dois exemplos:

1. Considere-se a função identidade $\lambda x.x$. $\forall_\varepsilon \forall_\rho [E\|\lambda x.x\|\rho \in T\|\alpha \rightarrow \alpha\|\varepsilon]$ e $\alpha \rightarrow \alpha$ é um tipo que pertence à linguagem de Curry. No entanto, também temos:

$$\forall_\varepsilon \forall_\rho [E\|\lambda x.x\|\rho \in T\|\alpha \rightarrow \beta\|\varepsilon]$$

   desde que $T\|\alpha\|\varepsilon \subseteq T\|\beta\|\varepsilon$. Neste caso $\alpha \rightarrow \beta$ tal que $T\|\alpha\|\varepsilon \subseteq T\|\beta\|\varepsilon$ não é exprimível na mesma linguagem pois não temos nenhum modo de denotar a inclusão de ideais. No entanto, existem ideais (todos os que respeitam a restrição $T\|\alpha\|\varepsilon \subseteq T\|\beta\|\varepsilon$) que denotam propriedades válidas da função identidade. Só que esses ideais não são exprimíveis na linguagem de tipos de Curry.

2. Considere-se o exemplo apresentado anteriormente da tipificação do termo $(\lambda x.xx)$ $(\lambda y.y)$. $\forall_\rho \forall_\varepsilon E\|(\lambda x.xx)(\lambda y.y)\|\rho = E\|\lambda y.y\|\rho \in T\|\alpha \rightarrow \alpha\|\varepsilon$. $\alpha \rightarrow \alpha$ é exprimível na linguagem de tipos de Curry, no entanto, o sistema de Curry não consegue relacionar este tipo ao termo $\lambda x.x$. Obter uma linguagem de tipos universal, isto é, que consiga exprimir todos os ideais possíveis é um objectivo irrealista e, além disso, completamente inútil, pois, nesse caso, as derivações aproximar-se-iam tanto dos próprios termos descritos que deixavam de fazer sentido. Defendemos que há uma extensão à linguagem de Curry que se justifica pela própria definição semântica dos tipos de Curry, que é a de permitir exprimir inclusão semântica de ideais na definição de novos ideais.

5.3 Restrições Semânticas

Recordemos a relação $A \models M : \tau$ apresentada no capítulo 2, com o significado de que τ é um tipo semanticamente admissível para M, se tivermos o conjunto de declarações para as variáveis livres A. A definição de $A \models M : \tau$ é:

$$A \models M : \tau \equiv \forall_\varepsilon \forall_\rho [\rho :_\varepsilon A \Rightarrow E\|M\|\rho \in T\|\tau\|\varepsilon]$$

Vamos analisar esta definição caso a caso, conforme a estrutura de M:

1. $M \equiv x$ em que x é uma variável. Neste caso, temos $\forall_\rho \forall_\varepsilon [\rho :_\varepsilon A \cup \{x : \sigma\} \Rightarrow \rho(x) \in T\|\tau\|\varepsilon]$. A única declaração de que depende o tipo de x é $\{x : \sigma\}$, portanto, a última equação toma a forma:

   $$\forall_\rho \forall_\varepsilon [\rho(x) \in T\|\sigma\|\varepsilon \Rightarrow \rho(x) \in T\|\tau\|\varepsilon]$$

   Isto implica:

   $$\forall_\varepsilon [T\|\sigma\|\varepsilon \subseteq T\|\tau\|\varepsilon]$$

   .

2. $M \equiv M_1 M_2$. Neste caso, temos $A \models M_1 M_2$ equivalente a

   $$\forall_\rho \forall_\varepsilon [\rho :_\varepsilon A \Rightarrow (E\|M_1\|\rho)|_F (E\|M_2\|\rho) \in T\|\tau\|\varepsilon]$$

   Isto significa que seja qual for ρ e ε, $\rho :_\varepsilon A$ implica que $E\|M_1\|\rho$ seja uma função que quando aplicada a $E\|M_2\|\rho$ fornece um elemento de $T\|\tau\|\varepsilon$. Seja $\alpha \to \beta$ o tipo de M_1 e σ o tipo de M_2. Então, a condição anterior é garantida desde que $T\|\sigma\|\varepsilon \in T\|\alpha\|\varepsilon \wedge T\|\beta\|\varepsilon \in T\|\tau\|\varepsilon$.

3. $M \equiv \lambda x.N$. Neste último caso temos $A \models \lambda x.N : \sigma$. $\forall_\rho \forall_\varepsilon [\rho :_\varepsilon A \Rightarrow \lambda v.(E\|N\|)$ $\rho[x/v] \in T\|\sigma\|\varepsilon]$. Isto significa que para todo o ρ e ε, $\rho :_\varepsilon A$ implica que $E\|\lambda x.N\|\rho$ seja uma função de x em $E\|N\|\rho$, onde ρ é o ambiente que a x associa o valor ao qual estamos a aplicar f. Se α for o tipo de x e β for o tipo de N, a condição anterior é verificada desde que $T\|\sigma\|\varepsilon$ contenha as funções de $T\|\alpha\|\varepsilon$ em $T\|\beta\|\varepsilon$, isto é, $T\|\alpha \to \beta\|\varepsilon \subseteq T\|\sigma\|\varepsilon$.

Como vimos, conforme os termos, temos um conjunto de restrições associadas que garantem a validade semântica do respectivo tipo.

5.4 Restrições

O nosso domínio de restrições é o conjunto de todos os ideais de V. As restrições são conseguidas à custa da inclusão de ideais. Sob o ponto de vista sintáctico, as restrições são inequações nos tipos, do género $\sigma_1 \sqsubseteq \sigma_2$. Assume-se que a relação $\sqsubseteq$ tem as seguintes propriedades:

1. É uma relação de ordem parcial;
2. Se $\sigma \to \tau \sqsubseteq \sigma' \to \tau'$, então $\sigma' \sqsubseteq \sigma$ e $\tau \sqsubseteq \tau'$.

5.5 O Sistema de Tipos com Restrições

A relação entre termos e tipos é apresentada por $A * R \vdash M : \tau$ onde A é uma base, M um termo, τ um tipo e R um conjunto de restrições, $R = \{\tau_1 \sqsubseteq \sigma_1 \wedge ... \wedge \tau_n \sqsubseteq \sigma_n\}$.

$$AXIOMA \quad A * R \cup \{\sigma \sqsubseteq \tau\} \vdash x : \tau \text{ ,se } x : \sigma \in A$$

$$\to ELIM \quad \frac{A * R_1 \vdash M : \alpha' \to \beta',\ A * R_2 \vdash N : \alpha}{A * R_1 \cup R_2 \cup \{\alpha \sqsubseteq \alpha', \beta' \sqsubseteq \beta\} \vdash MN : \beta}$$

$$\to INTRO \quad \frac{A \cup \{x : \alpha'\} * R \vdash M : \beta'}{A * R \cup \{\alpha \sqsubseteq \alpha', \beta' \sqsubseteq \beta\} \vdash \lambda x.M : \alpha \to \beta}$$

Vamos agora ver alguns exemplos:

1. Considere-se o termo $\lambda x.x$. A derivação de um tipo para este termo é dada pela seguinte árvore de derivação:

$$\begin{array}{c} \{x : \alpha'\} * \{\alpha' \sqsubseteq \beta'\} \vdash x : \beta' \\ \Big\downarrow {\scriptstyle ->INTRO} \\ \{\alpha' \sqsubseteq \beta', \alpha \sqsubseteq \alpha', \beta' \sqsubseteq \beta\} \vdash \lambda x.x : \alpha \to \beta \end{array}$$

Pela transitividade da relação $\sqsubseteq$ podemos derivar $\{\alpha \sqsubseteq \beta\} \vdash \lambda x.x : \alpha \to \beta$

2. Consideremos agora o termo $\lambda x.xx$, não tipificável no sistema de Curry. Neste sistema, derivamos $\{\sigma \sqsubseteq \alpha \to \beta, \sigma \sqsubseteq \alpha\} \vdash \lambda x.xx : \sigma \to \beta$. Seja $A = \{x : \sigma'\}$ e $R = \{\sigma' \sqsubseteq \alpha, \sigma' \sqsubseteq \alpha \to \beta'\}$. A árvore de derivação é a seguinte:

$$\begin{tikzcd}
A * R \vdash x : \alpha \to \beta' \arrow[dr] & A * R \vdash x : \alpha \arrow[d, "\to ELIM"'] \\
& A * R \vdash xx : \beta' \arrow[d, "\to INTRO"'] \\
& \{\sigma \sqsubseteq \sigma', \beta' \sqsubseteq \beta, \sigma' \sqsubseteq \alpha \to \beta', \sigma' \sqsubseteq \alpha\} \vdash \lambda x.xx : \sigma' \to \beta \arrow[d, "\text{transitividade de } \sqsubseteq"'] \\
& \{\sigma \sqsubseteq \alpha \to \beta, \sigma \sqsubseteq \alpha\} \vdash \lambda x.xx : \sigma \to \beta
\end{tikzcd}$$

5.5.1 Semântica

A relação semântica entre termos e tipos, é agora implicada, não só pelas declarações de tipos para as variáveis, mas também por restrições impostas aos tipos involvidos.

A avaliação de um conjunto de restrições com respeito a uma determinada atribuição de ideais é definida do seguinte modo:

Definição 5.1 *Dada uma atribuição ε e um conjunto de restrições R,*

$$R_\varepsilon \Leftrightarrow \forall_{\tau_1 \sqsubseteq \tau_2 \in R}[T\|\tau_1\|\varepsilon \subseteq T\|\tau_2\|\varepsilon]$$

Vamos, a partir de agora, e abusando da notação, assumir que, quando um conjunto de restrições R surgir na definição da relação $\models$, o seu significado é R_ε, ou seja, a sua avaliação com respeito a uma atribuição de tipos ε.

Definição 5.2 *Dada uma base A, um conjunto de restrições R, um termo M e um tipo τ:*

$$A * R \models M : \tau \Leftrightarrow \forall_\varepsilon \forall_{\rho \in Env}[\rho :_\varepsilon A \wedge R \Rightarrow E\|M\|\rho \in T\|\tau\|\varepsilon]$$

O sistema anterior é íntegro, isto é, $A * R \vdash M : \tau \Rightarrow A * R \models M : \tau$. Este resultado é apresentado no próximo teorema:

Teorema 5.1 *Sejam A uma base, M um termo, τ um tipo e R um conjunto de restrições, tais que $A * R \vdash M : \tau$. Então $A * R \models M : \tau$.*

Demonstração: Vamos usar indução estrutural em M.

1. Caso base: $M \equiv x$ (variável nos tipos). Neste caso $A * R \cup \{\sigma \sqsubseteq \tau\} \vdash x : \tau$, logo $\forall_\varepsilon[\{x : \sigma\} \in A \wedge T\|\sigma\|\varepsilon \subseteq T\|\tau\|\varepsilon]$. Pela definição de $\models$ temos $\forall_\rho\forall_\varepsilon[\rho(x) \in T\|\sigma\|\varepsilon \wedge T\|\sigma\|\varepsilon \subseteq T\|\tau\|\varepsilon \Rightarrow \rho(x) \in T\|\tau\|\varepsilon]$, logo $A * R \cup \{\sigma \sqsubseteq \tau\} \models x : \tau$.

2. $M \equiv M_1M_2$. $A * R_1 \cup R_2 \cup \{\alpha \sqsubseteq \alpha', \beta \sqsubseteq \tau\} \vdash M : \tau$. Pela hipótese de indução temos:

   (a) $A * R_1 \models M_1 : \alpha' \rightarrow \beta'$, o que é equivalente a

   $$\forall_\rho\forall_\varepsilon[\rho :_\varepsilon A \wedge R_1 \Rightarrow E\|M_1\|\rho \in T\|\alpha' \rightarrow \beta'\|\varepsilon]$$

   (b) $A * R_2 \models M_2 : \alpha$, o que é equivalente a

   $$\forall_\rho\forall_\varepsilon[\rho :_\varepsilon A \wedge R_2 \Rightarrow E\|M_2\|\rho \in T\|\alpha\|\varepsilon]$$

   Se tivermos como hipótese $R_3 = \{\alpha \sqsubseteq \alpha', \beta' \sqsubseteq \tau\}$, então temos:

   (a) $A * R_1 \cup R_3 \models M_1 : \alpha' \rightarrow \tau$ e

   (b) $A * R_1 \cup R_3 \models M_2 : \alpha'$

   o que implica, pela definição de $\models$, $A * R_1 \cup R_2 \cup R_3 \models M_1M_2 : \tau$.

3. $A * R \cup \{\alpha \sqsubseteq \alpha', \beta' \sqsubseteq \beta\} \vdash \lambda x.M : \alpha \rightarrow \beta$. Queremos demonstrar que $\forall_\varepsilon\forall_\rho[\rho :_\varepsilon A \Rightarrow E\|\lambda x.M\|\rho \in T\|\alpha \rightarrow \beta\|\varepsilon]$. Para isso, basta demonstrar que $\forall_\varepsilon\forall_\rho[\rho :_\varepsilon A \Rightarrow E\|M\|\rho[v/x]$ está em $T\|\beta\|\varepsilon$ para todo o v em $T\|\alpha\|\varepsilon]$.

   Pela hipótese de indução, sabemos que $A \cup \{x : \alpha'\} * R \models M : \beta'$, ou seja $\forall_\varepsilon\forall_\rho[\rho :_\varepsilon A \cup \{x : \alpha'\} \wedge R \Rightarrow E\|M\|\rho \in T\|\beta'\|\varepsilon]$, logo para todo o v em $T\|\alpha'\|\varepsilon$, $E\|M\|\rho[v/x]$ está em $T\|\beta'\|\varepsilon$. Mas ao assumirmos $\alpha \sqsubseteq \alpha'$ o mesmo é verdade para α, ou seja, para todo o v em $T\|\alpha\|\varepsilon$, $E\|M\|\rho[v/x]$ está em $T\|\beta'\|\varepsilon$. Como também temos como hipótese $\beta' \sqsubseteq \beta$, o mesmo se passa para $T\|\beta\|\varepsilon$, o que demonstra o pretendido. □

5.6 Resolução do Sistema de Restrições

Dada uma base A e um termo M, um tipo τ para M está associado a um conjunto R de restrições da forma $\tau_1 \sqsubseteq \tau_2$. De um modo geral, resolver um sistema de restrições deste género não é decidível, pois trata-se de um problema de semi-unificação, [KTU90]. No entanto, há subconjuntos de soluções para os quais temos algoritmos capazes de as determinar. A nossa tese é que definir um sistema de tipos não é mais do que identificar subconjuntos de soluções das restrições em causa. O conjunto de tipos que conseguimos inferir para um determinado termo é exactamente o subconjunto de soluções das restrições associadas à derivação desse termo que define o sistema de tipos. Neste capítulo, vamos começar por identificar um conjunto de soluções de R associado a uma derivação de $A \vdash M : \tau$. Esse conjunto, a que chamamos U é aquele que é possível determinar usando as soluções que satisfazem as restrições de igualdade. Se tivermos $A * R \vdash M : \tau$, onde cada elemento de R é da forma $\tau_1 \sqsubseteq \tau_2$, procuramos soluções do conjunto R' em que para cada $\tau_1 \sqsubseteq \tau_2 \in R$ temos $\tau_1 = \tau_2 \in R'$. Para isso, aplicamos o algoritmo de unificação de Robinson que dado R' retorna um conjunto de substituições u que aplicadas a τ e R retorna $A * u(R) \vdash u(\tau)$. Em $u(R)$ as restrições a que corresponde a unificação ficam da forma $u(\tau_1) = u(\tau_2)$ e as restantes permanecem sob a forma de inequações. As restrições resolvidas podem ser retiradas de $u(R)$ dando origem a um sistema de inequações estritas R_1. Ficamos assim com $A * R_1 \vdash u(\tau)$ em que R_1 é um conjunto de desigualdades estritas.

5.7 Relação com o Sistema de Curry

Comecemos pela seguinte definição:

Definição 5.3 *Um tipo de Curry, τ, diz-se próprio com respeito a uma base A e um termo M se e só se é possível têr $A \vdash M : \tau$, usando a unificação como estratégia de resolução de restrições.*

O conjunto de tipos próprios com respeito a uma base e a um termo M é o conjunto de tipos para os quais conseguimos resolver as restrições associadas, usando a unificação como método de resolução.

Se definirmos um novo sistema de tipos com restrições em que as restrições são equações em vez de inequações, temos trivialmente um sistema que só infere tipos próprios. Um sistema deste género é:

$$AXIOMA \quad A \vdash x : \tau \text{ ,se } \tau \in A$$

$$APL. \quad \frac{A \vdash M : \sigma \to \tau, A \vdash N : \sigma}{A \vdash MN : \tau}$$

$$ABS. \quad \frac{A \cup \{x : \sigma\} \vdash M : \tau}{A \vdash \lambda x.M : \sigma \to \tau}$$

Este sistema resultante da escolha das soluções respeitantes a igualdades no sistema de tipos com restrições é o sistema de Curry.

5.8 Discussão

Neste capítulo, apresentamos um novo sistema de tipos baseado directamente nas restrições semânticas impostas pela relação $\models$ apresentada no capítulo 2.

Este sistema, a que chamamos sistema de tipos com restrições, alarga o conjunto de termos tipificáveis e clarifica as relações semânticas impostas sobre os tipos. Além disso, dá uma visão de tipos como resultado da resolução de restrições. Cada sistema surge como um modo de resolver as restrições impostas e, o facto de um termo ter um determinado tipo num dado sistema equivale a conseguir resolver as restrições associadas. Se tomarmos equações como restrições, temos exactamente o sistema de Curry, o que clarifica e dá uma caracterização semântica deste sistema. O sistema de Curry surge, assim, a relacionar termos com os tipos resultantes de uma visão das restrições impostas pela relação semântica $\models$ correspondente à igualdade. Estas soluções são, de algum modo, mínimas, na medida em que, se tivermos restrições do género $\tau \sqsubseteq \alpha$, a solução mínima em α é exactamente τ. Dissemos "de algum modo", pois não podemos inferir exactamente soluções mínimas para um sistema de restrições somente à custa da igualdade. Podemos ter desigualdades estritas com soluções no domínio dos ideais, mas que não são exprimiveis por tipos de Curry, como é o caso do tipo de $\lambda x.xx$.

Salientamos o facto de existir, no conjunto de soluções descrito pelo sistema de Curry, algo que se prende com a essência do monomorfismo. De facto, ao exigirmos a satisfação de igualdades em restrições do género $\tau_1 \sqsubseteq \tau_2$, estamos a exigir a unicidade do tipo derivado para uma dada expressão. Se admitirmos derivações do género $A * R \vdash M : \tau$ onde as soluções de R representam conjuntos de tipos admissiveis para M, estamos a permitir uma representação polimórfica do tipo de M. Este carácter polimórfico das soluções de inequações estritas vai ser tornado claro nos capítulos 6 e 7, onde vamos estudar os conjuntos de soluções dos sistemas de restrições associados a sistemas que permitem polimorfismo, respectivamente, o sistema de Damas-Milner e o sistema de tipos com intersecções.

Finalmente, devemos referir a relação entre o sistema equivalente ao sistema de Curry e a unificação. É usando a unificação que resolvemos as restrições, que é exactamente o que acontece quando inferimos tipos no sistema de Curry. Resolução de restrições surge aqui ligada ao conceito de inferência de tipos dado que essa inferência é decidível no sistema em causa.

Capítulo 6

Polimorfismo

6.1 Introdução

Neste capítulo, aplicamos alguns resultados do capítulo anterior a sistemas de tipos que modelam polimorfismo paramétrico. O nosso objectivo é determinar de que modo os sistemas polimórficos podem ser representados por conjuntos de restrições e qual a estratégia de resolução de restrições seguida nestes sistemas.

O polimorfismo surge, no contexto de teoria de tipos, como a possibilidade de exprimir o facto de uma expressão ter mais do que um tipo. Um tipo polimórfico é uma representação de um conjunto de tipos, todos eles admissíveis para o mesmo termo. Esta última condição é essencial, na medida em que garante que o tipo polimórfico de um termo não representa tipos não admissíveis para esse termo. Introduzir este conceito num sistema de tipos passa por:

1. Permitir a derivação de tipos partindo do princípio que podemos ter mais do que um tipo para a mesma expressão, o que é conhecido por sobrecarga;
2. acrescentar à linguagem de tipos novos elementos sintácticos que representam conjuntos de tipos, como é o caso dos esquemas de tipos estudados no capítulo 3.

No segundo caso, temos necessariamente que acrescentar uma regra de generalização, isto é, uma regra que permita dados n tipos $\tau_1, ..., \tau_n$ tais que $A \vdash e : \tau_i$, $1 \leq i \leq n$ derivar a representação $\sigma = \{\tau_1, ..., \tau_n\}$, isto é, que garanta $A \vdash e : \sigma$. Temos também que garantir o inverso, isto é, se $A \vdash e : \sigma$ e $\sigma = \{\tau_1, ..., \tau_n\}$ então $A \vdash e : \tau_i$, $1 \leq i \leq n$. Como vamos ver uma escolha sensata da semântica dos tipos polimórficos faz com que esta última propriedade seja implícita ao sistema de tipos.

Como vimos no capítulo 3, a semântica de um tipo polimórfico σ é a intersecção das semânticas de todos os tipos que representa. Só assim garantimos que se um valor pertence a $T\|\sigma\|$ e $\sigma = \{\tau_1, ..., \tau_n\}$, então, pertence simultaneamente a $T\|\tau_i\|, 1 \leq i \leq n$. Logo, ao expandirmos o conceito de sistema de tipos com restrições ao polimorfismo temos $\sigma = \{\tau_1, ..., \tau_n\} \Rightarrow \sigma \sqsubseteq \tau_i, 1 \leq i \leq n$. O facto de termos que garantir que em qualquer sistema de tipos com polimorfismo $\sigma = \{\tau_1, ..., \tau_n\} \Rightarrow \sigma \sqsubseteq \tau_i, 1 \leq i \leq n$, faz com que, ao contrário do que acontece no capítulo anterior, a igualdade semântica não chegue para caracterizar o conjunto de tipos admissíveis para um dado termo. Estamos, assim, perante o facto de que o polimorfismo implica inclusão semântica estrita. Este ponto vai ser desenvolvido com detalhe no decorrer do capítulo. Vamos também demonstrar que uma escolha criteriosa de um subconjunto de soluções de um sistema polimórfico com restrições, modela o sistema de Damas-Milner, tal como acontecia no sistema de Curry.

Em resumo, os resultados fundamentais deste capítulo são:

1. A caracterização semântica do conceito de polimorfismo, que surge claramente como inclusão semântica estrita;

2. A demonstração de que o sistema de Damas-Milner respeita essa caracterização semântica e é, portanto, íntegro com relação a uma definição semântica das propriedades do polimorfismo em sistemas de tipos.

6.2 Polimorfismo nas Variáveis Livres

Em [Dam84] é apresentado um sistema de tipos que pressupõe tipos diferentes para as variáveis que ocorrem livres em determinados termos. Esse sistema justifica-se, pois serve como base a modelos de sobrecarga de operadores. No que diz respeito aos objectivos deste trabalho, ele representa uma primeira abordagem ao polimorfismo, pois permite às variáveis livres terem mais do que um tipo.

Como vimos no capítulo 3, uma nova expressão, o LET, é acrecentada ao λ-calculus para permitir tipificar expressões em que as abstracções são polimórficas, mas onde o seu contexto de ocorrência é limitado e conhecido. Assim, o sistema apresentado em [Dam84] é uma extensão ao sistema de Curry à custa de uma regra extra para tipificar as expressões LET onde podemos ter mais do que um tipo para as variáveis que ocorrem livres num termo. Note-se que ainda não falámos de esquemas de tipos, a linguagem de tipos ainda é limitada aos tipos de Curry. Neste contexto, vamos acrescentar a seguinte regra ao sistema com restrições definido no capítulo anterior:

$$LET \quad \frac{A * R_1 \vdash e : \sigma_1, ..., A * R_n \vdash e : \sigma_n, A \cup \{x : \tau_1, ..., x : \tau_n\} * R \vdash e' : \tau}{A * R \cup R_1 \cup, ..., \cup R_n \cup \{\sigma_1 \sqsubseteq \tau_1, ..., \sigma_m \sqsubseteq \tau_n, \tau \sqsubseteq \sigma\} \vdash \text{let } x = e \text{ in } e' : \sigma}$$

Acrescentando esta regra ao sistema com restrições simples, temos uma primeira aproximação ao polimorfismo considerando polimorfismo só nas variáveis livres nos antecedentes da regra LET. O que ganhamos com isso face ao sistema com restrições simples? Conseguimos resolver mais restrições do que o sistema anterior. Consideremos o seguinte exemplo:

Neste sistema, podemos resolver a restrição associada a let $x = \lambda y.y$ in xx que é semanticamente equivalente a $(\lambda x.xx)(\lambda y.y)$. Consideremos a derivação $\{x : \sigma \to \beta, x : \sigma\} \vdash xx : \beta\}$ e duas possíveis derivações de tipos para $\lambda y.y$, $\{\alpha \sqsubseteq \alpha'\} \vdash \lambda y.y : \alpha \to \alpha'$ e $\{\delta \sqsubseteq \delta'\} \vdash \lambda y.y : \delta \to \delta'$. Ao aplicar a regra LET ficamos com o seguinte conjunto de restrições:

$$\begin{cases} \alpha \to \alpha' & \sqsubseteq & \sigma \to \beta \\ \delta \to \delta' & \sqsubseteq & \sigma \\ \alpha & \sqsubseteq & \alpha' \\ \delta & \sqsubseteq & \delta' \end{cases}$$

Considerem-se apenas as equações respectivas. Após simplificações:

$$\begin{cases} \alpha \to \alpha & = & \sigma \to \beta \\ \delta \to \delta & = & \sigma \end{cases}$$

Uma solução deste sistema é:

$$\begin{array}{rcl} \sigma & = & \alpha_1 \to \alpha_1 \\ \beta & = & \alpha_1 \to \alpha_1 \\ \alpha & = & \alpha_1 \to \alpha_1 \\ \delta & = & \alpha_1 \end{array}$$

de onde derivamos $\vdash$ let $x = \lambda y.y$ in $xx : \alpha_1 \to \alpha_1$.

Este sistema, tal como o sistema simples com restrições é integro:

Teorema 6.1 *Dada uma base A, um termo M, um tipo τ e um conjunto de restrições R, $A * R \vdash M : \tau \Rightarrow A * R \models M : \tau$.*

Demonstração: Basta considerarmos o caso LET, pois as outras regras são semelhantes ao sistema do capítulo anterior.

Vamos relembrar que $E\|$let $x = e_1$ in $e_2\|\rho = E\|(\lambda x.e_2)e_1\|\rho$.. Por sua vez $E\|(\lambda x.e_2)e_1\|\rho = (E\|(\lambda x.e_2)\|\rho)|_F(E\|e_1\|\rho)$. Temos $A * R \cup R_1 \cup ... \cup R_n \cup \{\sigma_1 \sqsubseteq \tau_1, ..., \sigma_m \sqsubseteq \tau_n, \tau \sqsubseteq \sigma\} \vdash$ let $x = e_1$ in $e_2 : \sigma$. Pela hipótese de indução temos:

1. $A * R_i \models e_1 : \sigma_i, 1 \leq i \leq n$

2. $A \cup \{x : \tau_1, ..., x : \tau_n\} * R \models e_2 : \tau$

Vamos acrescentar às hipóteses as restrições $\forall_{1 \leq i \leq n}[\sigma_i \sqsubseteq \tau_i] \wedge \tau \sqsubseteq \sigma$.

Por 2. temos que $\forall_\varepsilon \forall_\rho [\rho :_\varepsilon A \cup \{x : \tau_1, ..., x : \tau_n\} \wedge R \Rightarrow E\|e_2\|\rho \in T\|\tau\|\varepsilon]$, ou seja, $\forall_\varepsilon \forall_\rho [\rho :_\varepsilon A \wedge \rho(x) \in T\|\tau_1\|\varepsilon \wedge ... \wedge \rho(x) \in T\|\tau_n\|\varepsilon \Rightarrow E\|e_2\|\rho \in T\|\tau\|\varepsilon]$, ou seja se $\rho(x)$ pertence a $T\|\tau_i\|\varepsilon \forall_{1 \leq i \leq n}$ então $E\|e_2\|\rho \in T\|\tau\|$. Isto significa que $E\|\lambda x.e_2\|\rho$ é uma função que quando aplicada a elementos de τ_i $1 \leq i \leq n$ fornece um elemento de τ. Mas por hipótese sabemos que $E\|e_1\|\rho$ está em σ_1 e σ_2 e ... e σ_n e cada $\sigma_i \sqsubseteq \tau_i$ $\forall_{1 \leq i \leq n}$. Logo $E\|e_1\|\rho$ está em τ_1 e τ_2 e...e τ_n. Sendo assim $E\|(\lambda x.e_2)e_1\|\rho$ está em $T\|\tau\|\varepsilon$ e consequentemente em qualquer ideal tal que $\tau \sqsubseteq \sigma$. Portanto, para quaisquer σ e σ_i, $1 \leq i \leq n$, tal que $\tau \sqsubseteq \sigma$ e $\sigma_1 \sqsubseteq \tau_i$, temos $\forall_\rho \forall_\varepsilon [\rho :_\varepsilon A \Rightarrow E\|(\lambda x.e_2)e_1\|\rho \in T\|\sigma\|\varepsilon]$ como queríamos demonstrar. □

Se considerarmos somente as soluções obtidas por unificação, que é equivalente a substituir no sistema todas as inequações por equações, obtemos o sistema descrito em [Dam84], o que demonstra que este último sistema, tal como o sistema de Curry, deriva exactamente os tipos próprios com relação a um determinado termo e a uma dada base.

6.3 Tipos Polimórficos

Como vimos no capítulo 3, o sistema de Damas-Milner aumenta a linguagem de tipos de modo a ter tipos polimórficos. Vamos ver como podemos modelar esta ideia usando restrições.

6.3.1 A Regra LET com Polimorfismo Explícito

A regra LET que vimos na secção anterior pressupõe uma forma de polimorfismo da variável x. Ao aumentarmos a linguagem de tipos de modo a ter tipos polimórficos, o facto de presupor na regra LET que x tem tipo τ_1 e τ_2 e ... e τ_n pode ser representado por um único tipo polimórfico $\mathcal{T}$ tal que se $\rho(x) \in T\|\mathcal{T}\|$ então $\rho(x) \in T\|\tau_1\| \wedge ... \wedge \rho(x) \in T\|\tau_n\|$, ou seja $\forall_{1\leq i \leq n} \mathcal{T} \sqsubseteq \tau_i$. A regra LET assumindo tipos polimórficos toma a forma:

$$LET \quad \frac{A * R_1 \vdash e : \mathcal{T}_1, A * R_2 \cup \{x : \mathcal{T}_2\} \vdash e' : \tau}{A * R_1 \cup R_2 \cup \{\mathcal{T}_1 \sqsubseteq \mathcal{T}_2, \tau \sqsubseteq \sigma\} \vdash \text{let } x = e \text{ in } e' : \sigma}$$

6.3.2 Tipos Próprios

Podemos ser levados a pensar que, tal como nos sistemas anteriores, usando somente as soluções das equações do sistema, temos uma caracterização deste tipo de polimorfismo. Tal não é verdade, pois como já foi referido, temos que ser capazes de inferir para x os tipos $\tau_1, ..., \tau_n$. Esta propriedade pressupõe que, implícita ou explicitamente, o sistema tenha que ter a seguinte propriedade:

$$INST \quad \frac{A \vdash e : \sigma, \sigma \sqsubseteq \sigma'}{A \vdash e : \sigma'}$$

Neste caso, σ e σ' podem ser tipos polimórficos.

Como vimos, é impossível, em sistemas com polimorfismo, resolver o conjunto de restrições associadas à derivação de tipos, usando apenas a unificação, ou seja, considerando apenas as soluções de equações. Temos que considerar soluções de desigualdades estritas para poder, a partir de um tipo polimórfico, chegar a cada uma das suas instâncias. Portanto, na presença de polimorfismo, temos que redefinir a noção de tipos próprios de modo a contemplar as instâncias de tipos polimórficos. Como vamos vêr, é possível expandir a definição de modo a abarcar só estes casos. A noção de tipo próprio permanece, assim, como a caracterização dos tipos que não são semanticamente redundantes em relação a um dado termo e a uma dada base.

Repare-se que a relação $\sqsubseteq$ definida como inclusão semântica contém outra relação $\sqsubseteq^*$, que representa os casos dos tipos representados por tipos polimórficos. Um tipo polimórfico é um tipo $\mathcal{T}$ que representa um conjunto T (finito ou infinito) de tipos.

Definição 6.1 *Dado um tipo polimórfico σ e o conjunto T_σ de tipos representado por σ definimos a relação $\sqsubseteq^*$ como: $\sigma \sqsubseteq^* \tau$ se e só se $\tau \in T_\sigma$.*

Um tipo polimórfico é uma representação de um conjunto de tipos. Se uma expressão e tem um tipo polimórfico σ, então, tem simultaneamente todos os tipos pertencentes a T_σ. Isto implica que $\sigma \sqsubseteq^* \tau \Rightarrow \sigma \sqsubseteq \tau$, pois $T\|\sigma\|\varepsilon = \bigcap_{\tau \in S_\tau} T\|\tau\|\varepsilon$. O inverso de um modo geral não se verifica, isto é, podemos ter tipos σ e τ tal que $\sigma \sqsubseteq \tau \wedge \sigma \not\sqsubseteq^* \tau$. Para verificar isto, basta aplicar a relação $\sqsubseteq$ a tipos simples.

Vamos agora redefinir a noção de tipo próprio:

Definição 6.2 *Os tipos próprios são aqueles que resultam de um sistema de tipos resultante da substituição no sistema com restrições correspondente de $\sqsubseteq$ por $=$, quando aplicado a tipos simples, e de $\sqsubseteq$ por $\sqsubseteq^*$, quando aplicado a tipos polimórficos.*

6.4 Generalização de Tipos

Qualquer sistema de tipos que tenha como objectivo a derivação de tipos polimórficos deve ter a propriedade de derivar todos os tipos polimórficos que seja possível exprimir na linguagem de tipos utilizada. De outro modo, poderíamos derivar um conjunto de tipos para um termo, esses tipos poderiam ser representados na linguagem usada por um tipo polimórfico que não era derivável, o que contradiz o objectivo destes sistemas. Tal como introduzimos a regra INST de modo a permitir chegar a cada tipo representado por um tipo polimórfico, temos agora que definir uma nova regra com o sentido inverso, isto é, que permita generalizar um conjunto de tipos derivados para um termo, desde que essa generalização seja exprimivel na linguagem de tipos.

Esta propriedade é expressa na seguinte regra de derivação:

$$POLIM \quad \frac{T = \{\tau_i | A * R_i \vdash e : \tau_i\}, \exists_\sigma . T_\sigma \subseteq T}{A * \bigcup_i R_i \vdash e : \sigma}$$

Se conseguirmos ter um tipo polimórfico σ que represente um conjunto de tipos de uma dada expressão e, então, podemos derivar σ para e. A exigência de que $T_\sigma \subseteq T$, onde T é o conjunto de todos os tipos de uma dada expressão, é necessária, pois, caso contrário podemos estar a considerar para e um tipo polimórfico que represente tipos que não são propriedades de e.

6.5 Polimorfismo e Restrições

Estamos agora em condições de definir o sistema de tipos polimórfico com restrições:

Seja A uma base, σ, σ', σ_1 e σ_2 tipos polimórficos ou simples, e todos os outros tipos usados na definição do sistema tipos simples.

$$AXIOMA \quad A * \{\sigma' \sqsubseteq \sigma\} \cup R \vdash x : \sigma, \text{ se } x : \sigma' \in A$$

$$\to ELIM \quad \frac{A * R_1 \vdash M_1 : \alpha' \to \beta', A * R_2 \vdash N : \alpha}{A * R_1 \cup R_2 \cup \{\alpha \sqsubseteq \alpha', \beta' \sqsubseteq \beta\} \vdash MN : \beta}$$

$$\to INTRO \quad \frac{A \cup \{x : \alpha'\} * R \vdash M : \beta'}{A * R \cup \{\alpha \sqsubseteq \alpha', \beta' \sqsubseteq \beta\} \vdash \lambda x.M : \alpha \to \beta}$$

$$LET \quad \frac{A * R_1 \vdash e : \sigma_1, A \cup \{x : \sigma_2\} * R_2 \vdash e' : \tau}{A * R_1 \cup R_2 \cup \{\sigma_1 \sqsubseteq \sigma_2, \tau \sqsubseteq \delta\} \vdash \text{let } x = e \text{ in } e' : \delta}$$

$$INST \quad \frac{A \vdash e : \sigma, \sigma \sqsubseteq \sigma'}{A \vdash e : \sigma'}$$

$$POLIM \quad \frac{T = \{\tau_i | A * R_i \vdash e : \tau_i\}, T_\sigma \subseteq T}{A * \bigcup_i R_i \vdash e : \sigma}$$

Uma propriedade esperada nestes sistemas é a seguinte: se, partindo da hipótese que uma dada variável x tem uma propriedade (tipo) τ, concluirmos que um termo tem a propriedade β, então, assumindo qualquer propriedade τ' que implique τ para x, continuamos a conlcuir que M tem a propriedade β. Isto é demonstrado no seguinte lema:

Lema 6.1 $A \cup \{x : \alpha\} * R \vdash M : \beta$ *e* $\alpha' \sqsubseteq \alpha \Rightarrow A \cup \{x : \alpha'\} * R \vdash M : \beta$

Demonstração: A demonstração torna-se trivial por indução estrutural em M, pois a hipótese é sobre uma declaração de tipos na base A que não muda após um passo de derivação. Vamos fazer em pormenor apenas o caso base em que M é uma variável x. Neste caso, temos $\{x : \alpha\} * \{\alpha \sqsubseteq \beta\} \vdash x : \beta$. Portanto, temos $\{x : \alpha'\} * \{\sigma \sqsubseteq \beta\} \vdash x : \beta$, pois, pela transitividade de $\sqsubseteq$, $\alpha' \sqsubseteq \alpha \Rightarrow [\alpha \sqsubseteq \beta \Rightarrow \alpha' \sqsubseteq \beta]$. □

A regra $INST$ pode ser retirada do sistema, pois é uma propriedade do sistema resultante como demonstramos no próximo teorema:

Teorema 6.2 *Dada uma base A, um termo M, dois tipos σ e σ' e um conjunto de restrições R tal que $A * R \vdash M : \sigma$ e $\sigma \sqsubseteq \sigma'$. Então $A * R \vdash e : \sigma'$*

Demonstração: Vamos usar indução no comprimento da árvore de derivação do tipo.

1. Caso base: $M \equiv x$. Este caso é trivial por aplicação directa da regra $AXIOMA$;

2. O termo é da forma $\lambda x.M$. Neste caso, partimos das seguintes hipóteses:

   (a) $A * R \cup \{\alpha \sqsubseteq \alpha', \beta' \sqsubseteq \beta\} \vdash \lambda x.M : \alpha \to \beta$;

   (b) $A \cup \{x : \alpha'\} * R \vdash M : \beta'$;

   (c) $A \cup \{x : \alpha'\} * R \vdash M : \tau, \forall_{\tau}.\beta' \sqsubseteq \tau$, por hipótese de indução.

   Dado que por hipótese temos $\{\beta' \sqsubseteq \beta\}$, podemos concluir que $\forall_{\beta^*}.\beta \sqsubseteq \beta^*$ temos $\beta' \sqsubseteq \beta^*$ pois $\beta' \sqsubseteq \beta$, logo $A \cup \{x : \alpha'\} \vdash M : \beta^*$ por c). $\alpha^* \sqsubseteq \alpha \Rightarrow \alpha^* \sqsubseteq \alpha'$ pois por hipótese, $\alpha \sqsubseteq \alpha'$, logo temos pelo lema 6.1, $\forall_{\alpha^* \sqsubseteq \alpha}[A \cup \{x : \alpha^*\} \vdash M : \beta^* \Rightarrow A \vdash \lambda x.M : \alpha^* \to \beta^*, \forall_{\alpha \to \beta \sqsubseteq \alpha^* \to \beta^*}$

3. $M \equiv M_1 M_2$. Neste caso $A \vdash M_1 M_2 : \beta$. Queremos demonstrar que $A \vdash M_1 M_2 : \beta' \, \forall_{\beta \sqsubseteq \beta'}$. Por hipótese sabemos que $A \vdash M_1 : \alpha' \to \beta^*$ se $\beta^* \sqsubseteq \beta$. Além disso $A \vdash M_1 : \alpha' \to \delta, \forall_{\beta^* \sqsubseteq \delta}$ (por hipótese de indução). Qualquer β' tal que $\beta \sqsubseteq \beta'$ é tal que $\beta^* \sqsubseteq \beta'$ porque $\beta^* \sqsubseteq \beta$ logo $A \vdash M_1 : \alpha' \to \beta'$. Pela regra $\to INTRO$ podemos concluir $A \vdash M_1 M_2 : \beta'$ como queríamos demonstrar.

4. $M \equiv let\ x = M_1 in\ M_2$. Neste caso temos $A \vdash let\ x = M_1 in\ M_2 : \delta \wedge \tau \sqsubseteq \delta$. Pela hipótese de indução sabemos que $A \cup \{x : \sigma\} \vdash M_2 : \tau' \forall_{\tau \sqsubseteq \tau'}$. Para um δ' arbitrário tal que $\delta \sqsubseteq \delta'$ temos $\tau \sqsubseteq \delta'$ (pois $\tau \sqsubseteq \delta \wedge \delta \sqsubseteq \delta'$). Aplicando a hipótese de indução e a regra LET concluimos $A \vdash let\ x = M_1 in\ M_2 : \delta'$ para um δ' arbitrário tal que $\delta \sqsubseteq \delta'$.

5. No caso de aplicarmos a regra $POLIM$, temos $A \vdash e : \sigma$, onde σ é um tipo polimórfico. Como σ é um tipo polimórfico, ele representa exactamente um conjunto de tipos T_σ, portanto, qualquer tipo τ tal que $\sigma \sqsubseteq \tau$ pertence a T_σ, e nesse caso, como $T_\sigma \in T$ onde T é o conjunto de todos os tipos deriváveis para e, temos $A \vdash e : \tau$. $\square$

Portanto, o sistema de tipos polimórfico com restrições fica agora reduzido às seguintes regras de derivação:

Seja A uma base, σ, σ', σ_1 e σ_2 tipos polimórficos ou simples, e todos os outros tipos usados na definição do sistema tipos simples.

$$AXIOMA \quad A * \{\sigma' \sqsubseteq \sigma\} \cup R \vdash x : \sigma, \text{ se } x : \sigma' \in A$$

$$\rightarrow ELIM \quad \frac{A * R_1 \vdash M_1 : \alpha' \rightarrow \beta', A * R_2 \vdash N : \alpha}{A * R_1 \cup R_2 \cup \{\alpha \sqsubseteq \alpha', \beta' \sqsubseteq \beta\} \vdash MN : \beta}$$

$$\rightarrow INTRO \quad \frac{A * R \cup \{x : \alpha'\} \vdash M : \beta'}{A * R \cup \{\alpha \sqsubseteq \alpha', \beta' \sqsubseteq \beta\} \vdash \lambda x.M : \alpha \rightarrow \beta}$$

$$LET \quad \frac{A * R_1 \vdash e : \sigma_1, A \cup \{x : \sigma_2\} * R_2 \vdash e' : \tau}{A * R_1 \cup R_2 \cup \{\sigma_1 \sqsubseteq \sigma_2, \tau \sqsubseteq \delta\} \vdash \text{let } x = e \text{ in } e' : \delta}$$

$$POLIM \quad \frac{T = \{\tau_i | A * R_i \vdash e : \tau_i\}, T_\sigma \subseteq T}{A * \bigcup_i R_i \vdash e : \sigma}$$

Vamos, agora, demonstrar a integridade do sistema:

Teorema 6.3 *Sejam A uma base, M um termo, σ um tipo (simples ou polimórfico) e R um conjunto de restrições, então, $A * R \vdash M : \sigma \Rightarrow A * R \models M : \sigma$.*

Demonstração: As regras $AXIOM$, $\rightarrow ELIM$ e $\rightarrow INTRO$ são tratadas da mesma forma que o sistema de tipos simples com restrições. Vamos, por isso, demonstrar a integridade do sistema para as regras POLIM e LET. A estratégia usada é indução no comprimento da árvore de derivação. O caso base é a aplicação da regra $AXIOMA$, semelhante à demonstração da integridade da mesma regra para o sistema simples com restrições. A integridade das regras $POLIM$ e LET é demonstrada a seguir:

1. $POLIM$: Neste caso temos $A * \bigcup R_i \vdash e : \sigma$. O nó anterior na árvore de derivação garante que $\forall_{\tau \in T_\sigma}[A * R_i \vdash e : \tau]$. Aplicando a hipótese de indução, temos $\forall_{\tau \in T_\sigma}[A * R_i \models e : \tau]$. Isto é equivalente a: $\forall_{\tau \in T_\tau} \forall_\rho \forall_\varepsilon [\rho :_\varepsilon A \wedge R_i \Rightarrow E\|e\|\rho \in T\|\tau\|\varepsilon]$, isto é, $\rho : A$ e R_i implicam que $E\|e\|\rho$ pertença a $T\|\tau\|\varepsilon, \forall_{\tau \in S_\sigma}$. Como $T\|\sigma\|\varepsilon = \bigcap_{\tau \in T_\sigma} T\|\tau\|\varepsilon$ isto implica que $\rho : A \wedge \bigcup R_i \Rightarrow E\|e\|\rho \in T\|\sigma\|\varepsilon$. Note-se que é fundamental considerar que $\forall_{\tau \in T_\sigma}[A * R_i \vdash e : \tau]$, pois, só assim, podemos aplicar a hipótese de indução e concluir que $\forall_{\tau \in T_\sigma}[A * \bigcup R_i \models e : \tau]$.

2. LET: Neste caso temos $A * R_1 \cup R_2 \cup \{\sigma \sqsubseteq \sigma', \tau \sqsubseteq \delta\} \vdash let\ x = M_1 in\ M_2 : \delta$. Por hipótese de indução, temos $A * R_1 \models M_1 : \sigma \wedge A \cup \{x : \sigma'\} * R_2 \models M_2 : \tau$.Vamos assumir $\sigma \sqsubseteq \sigma'$ e $\tau \sqsubseteq \delta$. Temos

$$\forall_\rho \forall_\varepsilon [\rho :_\varepsilon A \wedge R_1 \Rightarrow E\|M_1\|\rho \in T\|\sigma\|\varepsilon] \tag{6.1}$$
$$\forall_\rho \forall_\varepsilon [\rho :_\varepsilon A \wedge \rho(x) \in T\|\sigma'\| \wedge R_2 \Rightarrow E\|M_2\|\rho \in T\|\tau\|\varepsilon] \tag{6.2}$$

Por 6.2 concluimos que se $\rho(x) \in T\|\sigma'\|$ então $E\|M_2\|\rho \in T\|\tau\|\varepsilon$. Logo $E\|\lambda x.M_2\|\rho$ é uma função que quando aplicada a elementos de $T\|\sigma'\|\varepsilon$ retorna elementos de $T\|\tau\|\varepsilon$. Como $\forall_\varepsilon[T\|\sigma\|\varepsilon \subseteq T\|\sigma'\|\varepsilon]$, se aplicarmos $E\|\lambda x.M_2\|\rho$ a um elemento de $T\|\sigma\|\varepsilon$ obtemos um elemento de $T\|\tau\|\varepsilon$. Dado que $E\|M_1\|\rho \in T\|\sigma\|\varepsilon$ temos $E\|(\lambda x.M_2)M_1\|\rho \in T\|\tau\|\varepsilon$. Mas $T\|\tau\|\varepsilon \subseteq T\|\delta\|\varepsilon$, logo, podemos também concluir que $E\|(\lambda x.M_2)M_1\|\rho \in T\|\delta\|\varepsilon$. Atendendo a que $E\|let\ x = M_1 in\ M_2\|\rho = E\|(\lambda x.M_2)M_1\|\rho$ temos $\forall_\rho \forall_\varepsilon [\rho :_\varepsilon A \wedge R_1 \wedge R_2 \wedge \{\sigma \sqsubseteq \sigma', \tau \sqsubseteq \delta\} \Rightarrow E\|let\ x = M_1 in\ M_2\|\rho \in T\|\delta\|\varepsilon]$, o que é equivalente a $A * R_1 \cup R_2 \cup \{\sigma \sqsubseteq \sigma', \tau \sqsubseteq \delta\} \models let\ x = M_1 in\ M_2 : \delta$ □

6.6 Relação com o Sistema de Damas-Milner

Vamos definir um novo sistema onde derivamos exactamente os tipos próprios, isto é, substituimos, no sistema anterior, a relação $\sqsubseteq$ por $=$, quando aplicada a tipos simples e por $\sqsubseteq^*$ (tal como está definida em 7.2) quando aplicada a tipos polimórficos.

O sistema resultante, que denominamos sistema polimórfico próprio, é definido pelas seguintes regras de derivação:

$$AXIOMA \quad A * \{\sigma' \sqsubseteq^* \sigma\} \cup R \vdash x : \sigma, \text{ se } x : \sigma' \in A$$

$$\to ELIM \quad \frac{A * R \vdash M_1 : \alpha \to \beta, A * R \vdash N : \alpha}{A * R \vdash MN : \beta}$$

$$\to INTRO \quad \frac{A \cup \{x : \alpha'\} * R \vdash M : \beta}{A * R \vdash \lambda x.M : \alpha \to \beta}$$

$$LET \quad \frac{A * R_1 \vdash e : \sigma_1, A \cup \{x : \sigma_2\} * R_2 \vdash e' : \tau}{A * R_1 \cup R_2 \cup \{\sigma_1 \sqsubseteq^* \sigma_2, \tau \sqsubseteq^* \delta\} \vdash \text{let } x = e \text{ in } e' : \delta}$$

$$POLIM \quad \frac{T = \{\tau_i | A * R_i \vdash e : \tau_i\}, \exists_\sigma . T_\sigma \subseteq T}{A * \bigcup_i R_i \vdash e : \sigma}$$

Tal como acontecia em relação ao sistema de Curry, este sistema corresponde ao sistema de Damas-Milner, isto é, o sistema de Damas-Milner deriva para cada termo os tipos próprios com respeito a esse termo. Para demonstrar esse resultado, vamos, primeiro, referir uma propriedade do sistema de Damas-Milner que vai ser necessária à demonstração:

Teorema 6.4 *Seja* $T = \{\tau | A \vdash_{ML} M : \tau\}$*. Seja* π *o tipo principal de* M*. Então,* $\forall_\sigma [T_\sigma \subseteq T_\pi \Rightarrow \pi \sqsubseteq^* \sigma]$*.*

Demonstração: Vamos supor que existe um $\pi \not\sqsubseteq^* \sigma$ nas condições do teorema. Nesse caso, σ não é uma instância genérica de π. Então, existe uma instância de σ, τ, tal que $\tau \in T_\sigma$ e $\tau \notin T_\pi$. Mas isso contradiz a hipótese de que $T_\sigma \subseteq T_\pi$, de onde se demonstra o pretendido. □

Teorema 6.5 *Seja* A *uma base,* M *uma expressão,* σ *um tipo e* R *um conjunto de restrições.* $A * R \vdash M : \sigma$ *tem como significado que derivamos* σ *para* M *partindo de* A *no sistema polimórfico próprio e* $A \vdash_{ML} M : \sigma$ *significa que derivamos* σ *para* M *a* *partir de A no sistema de Damas-Milner. Se o conjunto de restrições R se verificar então:*

$$A \vdash M : \sigma \Leftrightarrow A \vdash_{ML} M : \sigma$$

Demonstração: Vamos separar a demonstração do teorema em duas partes; uma que demonstra $A \vdash_{ML} e : \sigma \Rightarrow A \vdash e : \sigma$ e a outra $A \vdash e : \sigma \Rightarrow A \vdash_{ML} e : \sigma$. Em qualquer dos casos, vamos usar indução no comprimento da árvore de derivação dos tipos.

Caso 1: $A \vdash_{ML} e : \sigma \Rightarrow A \vdash e : \sigma$:

1. Caso base: a regra aplicada é o $AXIOMA_{ML}$. $A \vdash_{ML} x : \sigma$ se $\{x : \sigma\} \in A$ portanto $A \vdash e : \sigma$;
2. Os casos respeitantes às regras $\rightarrow ELIM_{ML}$ e $\rightarrow INTRO_{ML}$ são triviais por comparação directa das regras respectivas nos dois sistemas;
3. No caso LET_{ML} basta considerarmos as soluções respeitantes às igualdades no sistema próprio para concluirmos o pretendido;
4. Vamos agora supor que a última regra aplicada é a instância($INST$). Sabemos por hipótese de indução que $A \vdash e : \sigma$ e queremos demonstrar que $A \vdash e : \sigma' \forall \sigma' [\sigma \sqsubseteq^* \sigma]$. Isto é verdade pelo teorema 6.2.
5. Generalização (aplicação da regra GEN): Sabemos por hipótese de indução que $A \vdash e : \tau \; \forall_{\tau \in C} \wedge C \subseteq T_\sigma$. Sabemos também que existe σ tal que $A \vdash_{ML} e : \sigma$ e $A \vdash_{ML} e : \tau' \forall_{\tau'} [\sigma \sqsubseteq^* \tau']$, pela regra de $INST$. Como vimos pelos casos anteriores $A \vdash_{ML} e : \tau'$ e τ' é um tipo simples implica $A \vdash e : \tau'$. Logo $A \vdash e : \tau' \forall_{\tau'} \in T_\sigma$ e $T_\sigma \subseteq T$, sendo $T = \{\tau_i | A \vdash e : \tau_i\}$. Pela regra de POLIM temos $A \vdash e : \sigma$ como queríamos demonstrar.

Caso 2: $A \vdash e : \sigma \Rightarrow A \vdash_{ML} e : \sigma$:

1. Caso base ($AXIOMA$): Neste caso, temos $A \vdash x : \sigma' \Rightarrow x : \sigma \in A \wedge \sigma \sqsubseteq^* \sigma'$. Pela regra do $AXIOMA_{ML}$ e pela regra de $INST$, concluimos que $A \vdash_{ML} x : \sigma'$;
2. Se a última regra aplicada for $\rightarrow ELIM$ ou $\rightarrow INTRO$, o resultado é trivial por comparação das regras respectivas nos dois sistemas;
3. LET: Temos $A \vdash \mathit{let}\ x = M_1 \mathit{in}\ M_2 : \tau$. Pela hipótese de indução, temos $A \vdash_{ML} M_1 : \sigma \wedge A \cup \{x : \sigma'\} \vdash_{ML} M_2 : \tau \wedge \sigma \sqsubseteq^* \sigma'$. Pela regra de $INST$, temos $A \vdash M_1 : \sigma'$ e pela regra LET_{ML} temos $A \vdash_{ML} \mathit{let}\ x = M_1 \mathit{in}\ M_2 : \tau$;

4. $POLIM$: Pela propriedade do tipo principal do sistema de Damas-Milner, temos $T_\pi = \{\tau | A \vdash_{ML} e : \tau\} \Rightarrow A \vdash_{ML} e : \pi$. Pela definição da regra POLIM, temos $A \vdash e : \sigma \Rightarrow A \vdash e : \tau \forall_{\tau \in T_\sigma}$ e $T_\sigma \subseteq T_\pi$. Por hipótese de indução $A \vdash_{ML} e : \tau \forall_{\tau \in T_\sigma}$ e $T_\sigma \subseteq T_\pi$. Pelo teorema anterior, temos $\pi \sqsubseteq^* \sigma$, logo pela regra de $INST_{ML}$ temos $A \vdash_{ML} e : \sigma$ $\Box$

6.7 Discussão

Neste capítulo estendemos os resultados obtidos no capítulo anterior a linguagens de tipos que permitam expressar polimorfismo. Demonstrámos, essencialmente, o seguinte:

1. O conceito de polimorfismo está ligado à inclusão semântica estrita, não sendo possível, só à custa da resolução de restrições semânticas expressas por equações derivar tipos polimórficos;
2. O sistema de Damas-Milner, que está na base do sistema de tipos da linguagem de programação ML, deriva exactamente os tipos próprios com respeito a cada termo. O modo como definimos tipos próprios garante que o sistema em causa não deriva tipos redundantes sob o ponto de vista semântico.

O nosso objectivo com a elaboração do sistema polimórfico com restrições não foi a inferência automática de tipos. Isso não era de esperar, pois o nosso sistema assenta directamente na relação semântica entre termos e tipos, que não é decidível, e é definido à custa de sistemas de inequações, cuja resolução geral passa por semi-unificações, que também não são decidíveis. O objectivo foi conseguir um sistema polimórfico geral que, tal como o sistema simples, fosse exprimível à custa de conjuntos restrições nos tipos admissíveis para cada termo. Mostrámos as potencialidades do sistema para o caso decidível (sistema de Damas-Milner). Pensamos que um trabalho semelhante para sistema polimórficos não decidíveis, como o sistema de Mycroft-Milner [Myc84] ou o sistema F [Gir72], deveria levar a conclusões interessantes sobre a não decidibilidade desses sistemas. No entanto, isso ficará para trabalho futuro.

Em teoria de tipos, existem outros contextos onde foram usados sistemas de inequações para o estudo de propriedades dos sistemas. Em [Hen93], a derivação de tipos no sistema de Mycroft-Milner é reduzida ao problema da semi-unificação para demonstrar a não decidibilidade do sistema em causa. Em [EST95] e [PS96], são definidos sistemas que usam restrições para modelar tipos em linguagens orientadas por objectos. Nós usamos restrições com outros objectivos, pois:

1. No nosso sistema, as restrições são consequência de restrições semânticas nos ideais impostas pela tipificação semântica (relação $\models$).

2. Usamos conjuntos de restrições para exprimir os tipos admissíveis sob o ponto de vista semântico e mostramos que os sistemas de tipos tradicionais não são mais do que resolutores de restrições.

Este trabalho tem aplicações que deixamos para trabalho futuro. As mais interessantes na nossa opinião são:

1. Aplicação de técnicas de resolução de restrições à inferência de tipos. Esta ideia passa por um módulo de definição de restrições semânticas e, posteriormente, pela resolução de algumas dessas restrições em resolutores correspondentes aos vários sistemas de tipos;

2. Aplicação de técnicas baseadas no atraso da resolução das restrições que o resolutor não consegue resolver até se tornarem suficientemente simples para serem resolvidas. Esta técnica é usada em módulos de restrições em linguagens de programação por restrições ([Flo91] e [JJ87]) e parece-nos um modo de obter inferência de tipos para sistema não decidíveis.

Capítulo 7

Integridade nos Sistemas de Tipos com Intersecções

7.1 Introdução

Neste capítulo, analisamos um problema de alguns sistemas de tipos com intersecções, identificado por Van Bakel em [Bak93a] que é a falta de integridade da noção de instância. Este problema reside no seguinte: podemos ter $A \vdash M : \sigma$, τ uma instância de σ e $A \not\vdash M : \tau$. A falta de integridade da noção de instância torna impraticável o uso de tipos principais para verificação de tipos, pois nada garante que o facto de um tipo ser uma instância do tipo principal o torne um tipo admissível para um determinado termo. Existem extensões que, com uma regra:

$$(\leq) \quad \frac{A \vdash M : \sigma, \sigma \leq \tau}{A \vdash M : \tau}$$

garantem que qualquer instância de um tipo que tipifica um determinado termo tipifica o mesmo termo.

Nós resolvemos este problema definindo um novo sistema de tipos onde existe uma regra de instância explícita, limitada às variáveis, do tipo:

$$(\sqsubseteq) \quad \frac{A \vdash x : \sigma, \sigma \sqsubseteq \tau}{A \vdash x : \tau}$$

e justificamo-lo, fazendo uma análise baseda em restrições induzidas pela relação semântica entre termos e tipos com intersecções, usando uma abordagem semelhante à dos últimos capítulos.

O sistema em causa é o sistema de Coppo-Dezani [CD80]. O problema focado pode ser ilustrado com o seguinte exemplo apresentado em [Bak93b]:

Exemplo 7.1 $\vdash \lambda x.x : (\alpha \to \beta) \to \alpha \to \beta$. *Pela definição de instância neste sistema,* $(\alpha \to \beta) \to \alpha \to \beta \sqsubseteq (\alpha \to \beta) \to \alpha \cap \sigma \to \beta$ *e* $\not\vdash \lambda x.x : (\alpha \to \beta) \to \alpha \cap \sigma \to \beta$.

Poderíamos ser levados a pensar que o problema está no facto da noção de instância ser demasiado "permissiva". Tal não é o caso. Como mostra o próximo exemplo, retirado de [Bak93b], no caso anterior temos que ter $(\alpha \to \beta) \to \alpha \to \beta \sqsubseteq (\alpha \to \beta) \to \alpha \cap \sigma \to \beta$ se quisermos ter uma noção de instância que englobe certo tipo de derivações.

Exemplo 7.2 *Considerem-se as derivações para* $\vdash \lambda xy.xy : (\alpha \to \beta) \to \alpha \to \beta$ *e* $\vdash \lambda xy.xy : (\alpha \to \beta) \to \alpha \cap \sigma \to \beta$*:*

$$\begin{array}{rcl}
\{x : \alpha \to \beta, y : \alpha\} & \vdash & xy : \beta \\
\{x : \alpha \to \beta\} & \vdash & \lambda y.xy : \alpha \to \beta \\
 & \vdash & \lambda xy.xy : (\alpha \to \beta) \to \alpha \to \beta
\end{array}$$

$$\begin{array}{rcl}
\{x : \alpha \to \beta, y : \alpha \cap \sigma\} & \vdash & y : \alpha \\
\{x : \alpha \to \beta, y : \alpha \cap \sigma\} & \vdash & xy : \beta \\
\{x : \alpha \to \beta\} & \vdash & \lambda y.xy : \alpha \cap \sigma \to \beta \\
 & \vdash & \lambda xy.xy : (\alpha \to \beta) \to \alpha \cap \sigma \to \beta
\end{array}$$

Para passar de $(\alpha \to \beta) \to \alpha \to \beta$ para $(\alpha \to \beta) \to \alpha \cap \sigma \to \beta$ temos que ter uma operação tal que $(\alpha \to \beta) \to \alpha \to \beta \sqsubseteq (\alpha \to \beta) \to \alpha \cap \sigma \to \beta$. Portanto, o problema não está na noção de instância, mas sim no sistema de tipos.

Nós resolvemos este problema introduzindo uma nova regra, semelhante à regra de instância do sistema de Damas-Milner, do género:

$$(\sqsubseteq) \quad \frac{A \vdash x : \sigma, \sigma \sqsubseteq \tau}{A \vdash x : \tau}$$

Estes resultados demonstram que a simplificação ao sistema de Damas-Milner ao retirar a regra de instância é falsa, pois, sem uma regra semelhante, perdemos a integridade dos tipos principais.

Demonstramos que este novo sistema tem a propriedade do tipo principal e a mesma expressividade do sistema de Coppo-Dezanni, isto é, tipifica exactamente os termos fortemente normalizáveis.

Os resultados negativos do estudo dos sistemas com intersecções, levaram-nos a ver com detalhe qual o comportamento destes sistemas quando modelados por restrições semânticas, como foi feito para os sistemas de Curry e de Damas-Milner. Um estudo semelhante ao efectuado nos capítulos anteriores levou a duas conclusões fundamentais:

1. Existem tipos próprios com respeito a um dado termo que o sistema de Coppo-Dezanni não deriva. É o caso de $(\alpha \to \beta) \to \alpha \cap \sigma \to \beta$ que é próprio com respeito a $\lambda x.x$ e $\nvdash \lambda x.x : (\alpha \to \beta) \to \alpha \cap \sigma \to \beta$. Este problema é semelhante ao problema da integridade da noção de instância, se virmos as instâncias de um determinado tipo que o sistema de Coppo-Dezanni não deriva, como os tipos próprios que o mesmo sistema não consegue derivar;

2. Ao resolver o problema anterior, temos que alterar o sistema com intersecções de modo a derivar todos os tipos próprios com respeito a um determinado termo. Essa alteração é a mesma que efectuámos para resolver o problema da falta de integridade da noção de instância. Isto dá uma justificação sob o ponto de vista semântico à alteração feita inicialmente ao sistema de Coppo-Dezanni.

No decorrer do capítulo, começamos por definir o novo sistema de tipos com intersecções. Em seguida, definimos um sistema de tipos com restrições, aumentando com intersecções a linguagem de tipos e mostramos como, a partir de uma análise do conjunto de soluções de restrições que obtemos usando o sistema de Coppo-Dezani, chegamos inevitavelmente ao problema inicial deste sistema: a falta de integridade da noção de instância. Finalmente, justificamos a alteração feita no inicio ao sistema original como sendo indispensável para a derivação de tipos próprios em sistemas com intersecções.

7.2 Um Novo Sistema de Tipos com Intersecções

Tendo como objectivo o estudo de sistemas de tipos em que o argumento de uma função possa ter mais do que um tipo, iniciou-se um estudo do sistema de Coppo-Dezani, pois a extensão a este sistema que acrescenta o tipo universal ω não nos parece relevante tendo em conta o objectivo inicial de permitir polimorfismo no argumento das funções. Este estudo levou-nos à criação de um novo sistema de tipos com o objectivo de superar algumas falhas detectadas no sistema original. Começamos por estudar para o sistema de Coppo-Dezani qual a relação entre os diversos tipos admissíveis. Para isso, definiu-se uma relação de ordem entre os diversos tipos que capture, no contexto de tipos com intersecção, a noção de instância, isto é, $\tau_1 \sqsubseteq^* \tau_2$ se e só se τ_2 for uma instância de τ_1.

7.2.1 Instância

Em sistemas de tipos em que a única operação de transformação de tipos é a substituição, a noção de instância surge à custa da substituição. Para estes sistemas τ_1 é uma instância de τ_2 se existir uma substituição s tal que $s(\tau_2) = \tau_1$.

Em sistemas com intersecções, o correspondente à substituição são cadeias. Recorde-se que uma cadeia c é uma composição de susbtituições, expansões e elevações. Portanto, a noção de instância é definida por uma relação $\sqsubseteq^*$ entre tipos do seguinte modo:

Definição 7.1 *Sejam σ_1 e σ_2 dois tipos. $\sigma_1 \sqsubseteq^* \sigma_2$ se e só se existe uma cadeia c tal que $c(\sigma_1) = \sigma_2$.*

É fácil mostrar que se trata de uma pré-ordem.

Teorema 7.1 *Sejam σ_1 e σ_2 dois tipos. Seja $\sqsubseteq^*$ uma relação tal que $\sigma_1 \sqsubseteq^* \sigma_2$ se e só se existe uma cadeia c tal que $c(\sigma_1) = \sigma_2$. Então $\sqsubseteq^*$ é uma pré-ordem.*

Demonstração:

1. $\sqsubseteq^*$ é reflexiva. Basta considerar a cadeia vazia.

2. $\sqsubseteq^*$ é transitiva.

$$\sigma_1 \sqsubseteq^* \sigma_2 \Rightarrow \exists_{c_1}.c_1(\sigma_1) = \sigma_2$$

$$\sigma_2 \sqsubseteq^* \sigma_3 \Rightarrow \exists_{c_2}.c_2(\sigma_1) = \sigma_3$$

Considere-se a composição $c_2 \circ c_1$. Então $c_2 \circ c_1(\sigma_1) = \sigma_2 \Rightarrow \sigma_1 \sqsubseteq^* \sigma_3$. □

Esta relação induz, no conjunto, uma relação de equivalência: $\sigma_1 \equiv \sigma_2$ se e só se $\sigma_1 \sqsubseteq^* \sigma_2$ e $\sigma_2 \sqsubseteq^* \sigma_1$. Como se demonstra no próximo teorema, considerando o conjunto de classes de equivalência T módulo $\equiv$, ordenado pela relação $\sqsubseteq^*$, obtemos um conjunto parcialmente ordenado com elemento mínimo α.

Teorema 7.2 *O conjunto de tipos T juntamente com a relação $\sqsubseteq^*$ forma um conjunto parcialmente ordenado Γ, tendo como elemento mínimo a classe das variáveis α.*

Demonstração: Nesta demonstração, usamos indução na estrutura dos tipos.

1. *Caso Base:* Pela reflexividade de $\sqsubseteq^*$ qualquer variável é tal que $\alpha \sqsubseteq^* \alpha$.

2. *Passo de Indução:* $\bar{\sigma_1} \to \sigma_2$. Se σ_1 não contiver intersecções $\bar{\sigma_1} \to \sigma_2$ é obtido de α trivialmente por substituição. Se $\bar{\sigma_1}$ for da forma $\tau_1 \cap \ldots \cap \tau_n$ substituimos α por $\alpha \to \sigma_2$, em seguida expandimos α para $\alpha_1 \cap \ldots \cap \alpha_n$ e, pela hipótese de indução, temos cadeias $c_1, \ldots, c_2$ que transformam uma variável em $\tau_1, \ldots, \tau_n$, respectivamente. Logo $c_n \circ \ldots \circ c_1(\alpha_1 \cap \ldots \cap \alpha_n \to \sigma_2) = \tau_1 \cap \ldots \cap \tau_n \to \sigma_2$ □

Um fragmento da representação gráfica de Γ é:

$$\begin{tikzcd}
 & ((\alpha_1 \cap \alpha_2) \to (\alpha_3 \cap \alpha_4)) \to \beta \arrow[d] \arrow[dr] & \\
\alpha \to \alpha \arrow[dr] & \alpha_1 \cap \alpha_2 \to \beta \arrow[d] & (\alpha_1 \to \alpha_2) \to \beta \arrow[dl] \\
 & \alpha \to \beta \arrow[d] & \\
 & \alpha &
\end{tikzcd}$$

Uma outra propriedade da relação $\sqsubseteq^*$ é a de que, se tivermos uma sequência $\sigma_1 \cap \ldots \cap \sigma_n$ e lhe aplicarmos uma cadeia c tal que $c(\sigma_1 \cap \ldots \cap \sigma_n) = \tau$ e τ é um tipo, então $\exists_{\sigma_i}[\sigma_i \sqsubseteq^* \tau]$.

Lema 7.1 *Seja $\bar{\sigma}$ uma sequência. Então $\forall_\tau.\bar{\sigma} \sqsubseteq \tau$ τ é um tipo $\Rightarrow \exists_{\sigma_i \in \sigma}.\sigma_i \sqsubseteq^* \tau$.*

Demonstração: Vamos usar indução no comprimento da cadeia c tal que $c(\bar{\sigma}) = \tau$.

Caso base: o caso base corresponde a c ser uma substituição, uma expansão ou uma elevação. Vamos ver cada um dos casos:

1. Substituição: se $\bar{\sigma}$ for uma sequência com tamanho maior do que 1, τ não pode ser um tipo pela definição de substituição e de tipo;

2. Expansão: o mesmo acontece com a expansão pois a expansão de uma intersecção é a intersecção das expansões de cada elemento;

3. Elevação: neste caso, chegamos de uma sequência a um tipo, se esse tipo pertencer à sequência, pela definição de elevação quando aplicada a sequências;

Passo de indução: Se tivermos uma cadeia de tamanho n que transforma $\bar{\sigma}$ num tipo τ' e, por hipótese de indução, τ' resultar de um elemento de $\bar{\sigma}$, então, ao aumentar o tamanho da cadeia para $n+1$ para obtêr τ partimos do mesmo elemento de $\bar{\sigma}$. $\Box$

A relação $\sqsubseteq^*$ é extensível a bases: $B_1 \sqsubseteq^* B_2$ se e só se $\forall_x[x : \tau_1 \in B_1 \wedge x : \tau_2 \in B_2 \Rightarrow \tau_1 \sqsubseteq^* \tau_2]$.

7.2.2 Integridade da Noção de Instância

Nesta fase, deparamos com um problema no sistema de Coppo-Dezani. Como foi observado em [Bak93b], a operação de instância não é íntegra para o sistema de Coppo-Dezani com o tipo universal ω. Este resultado negativo também se aplica para o sistema de Coppo-Dezani. De facto $B_1 \vdash M : \sigma_1$ e $< B_1, \sigma_1 > \sqsubseteq^* < B_2, \sigma_2 >$ não garante que $B_2 \vdash M : \sigma_2$ se verifique. O problema reside na operação de elevação. O exemplo apresentado em [Bak93b] serve para demonstrar essa falha também no sistema de Coppo-Dezani.

Exemplo 7.3 $\vdash \lambda x.x : (\alpha \to \beta) \to \alpha \to \beta$. *Pela definição de elevação* $(\alpha \to \beta) \to \alpha \to \beta \sqsubseteq^* (\alpha \to \beta) \to \alpha \cap \sigma \to \beta$ *e* $\not\vdash \lambda x.x : (\alpha \to \beta) \to \alpha \cap \sigma \to \beta$.

Um problema semelhante surgiu quando da introdução de polimorfismo paramétrico. Ao permitir generalizar um conjunto de tipos através de um esquema de tipos, temos que garantir que, se inferimos um esquema de tipos para dada expressão, qualquer instância genérica é também inferível para a mesma expressão. Isto consegue-se através da regra de instância:

$$\frac{A \vdash M : \forall_\alpha.\sigma}{A \vdash M : \sigma[\alpha/\tau]}$$

Como vamos ver, basta acrescentar uma regra semelhante à instância do polimorfismo paramétrico nas variáveis para obter integridade de $\sqsubseteq^*$ em relação à inferência de tipos.

O novo sistema é igual ao sistema de Coppo-Dezani mais uma regra $\sqsubseteq^*$ que permite a inferência de instâncias de qualquer tipo inferido para uma variável. As regras de inferência são:

$$AXIOM \quad A \vdash x : \sigma_i \text{ ,se } \sigma_i \in \bar{\sigma} \text{ e } x : \bar{\sigma} \in A$$

$$(\sqsubseteq^*) \quad \frac{A \vdash x : \sigma, \sigma \sqsubseteq^* \tau}{A \vdash x : \tau}$$

$$\to ELIM \quad \frac{A \vdash M : \sigma_1 \cap \ldots \cap \sigma_n \to \sigma, A \vdash N : \sigma_1, \ldots, A \vdash N : \sigma_n}{A \vdash MN : \sigma}$$

$$\to INTRO \quad \frac{A \cup \{x : \sigma_1 \cap \ldots \cap \sigma_n\} \vdash M : \sigma}{A \vdash \lambda x.M : \sigma_1 \cap \ldots \cap \sigma_n \to \sigma} \quad (a)$$

(a) Neste caso $x : \sigma_1 \cap \ldots \cap \sigma_n$ é a única declaração para x da qual depende $M : \sigma$.

Vamos agora apresentar algumas propriedades importantes do novo sistema de tipos:

Lema 7.2 [Geração de tipos]

1. $A \vdash x : \sigma \Rightarrow \exists_{\bar{\theta}}[x : \bar{\theta} \in A \wedge \bar{\theta} \sqsubseteq^* \sigma]$*;*

2. $A \vdash MN : \tau \Rightarrow \exists_{\bar{\sigma}}[A \vdash MN : \bar{\sigma} \to \tau \wedge A \vdash N : \sigma_i \forall_{\sigma_i \in \bar{\sigma}}$*;*

3. $A \vdash \lambda x.M : \rho \Rightarrow \exists_{\bar{\sigma},\tau}[A \cup \{x : \bar{\sigma}\} \vdash M : \tau \wedge \rho \equiv \sigma \to \tau]$

Demonstração: Usa-se indução no comprimento das derivações.

1. *Caso base:* a derivação tem comprimento 1. Neste caso, aplicamos a regra do AXIOMA e 1) verifica-se trivialmente.

2. Passo de indução: se o termo é uma variável e temos $n-1$, com $n > 2$, passos de indução, a última regra de inferência aplicada é $\sqsubseteq^*$, o que mostra 1). Se o termo for da forma MN, a última regra de inferência aplicada é $\rightarrow ELIM$ o que demonstra 2). Finalmente, se o termo é da forma $\lambda x.M$ a última regra aplicada é $\rightarrow$INTRO , logo 3) verifica-se. □

Lema 7.3 *Se $A \cup \{x : \bar{\sigma'}\} \vdash M : \tau$ e $\bar{\sigma} \sqsubseteq^* \bar{\sigma'}$ então $A \cup \{x : \bar{\sigma}\} \vdash M : \tau$.*

Demonstração: Usa-se indução estrutural em M.

1. *Caso base, $M \equiv x$.* $A \cup \{x : \bar{\sigma'}\} \vdash x : \tau \Rightarrow \bar{\sigma'} \sqsubseteq^* \tau$. Pela transitividade de $\sqsubseteq^*$ temos $\bar{\sigma} \sqsubseteq^* \bar{\sigma'} \sqsubseteq^* \tau \Rightarrow \bar{\sigma} \sqsubseteq^* \tau$ logo $A \cup \{x : \bar{\sigma}\} \vdash x : \tau$.

2. *Passos de indução:*

   (a) *$M \equiv \lambda y.M'$:* Se $y \neq x$ basta aplicar a hipótese de indução e a regra de $\rightarrow$ INTRO. Se $y = x$ então y não ocorre livre em M', logo as hipótese em y não influenciam a derivação do tipo τ para M'. Portanto $\forall_{\bar{\theta}}[A \cup \{y : \bar{\theta}\} \vdash M : \tau \forall_{\bar{\theta}} \Rightarrow A \vdash \lambda y.M : \bar{\theta} \rightarrow \tau]$.

   (b) *$M \equiv M_1 M_2$:* Pelo lema 7.2 $\exists_{\bar{\theta}}[A \cup \{x : \sigma'\} \vdash M_1 : \bar{\theta} \rightarrow \tau]$ e $\forall_{\theta_i \in \bar{\theta}}[A \cup \{x : \sigma'\} \vdash M_2 : \theta_i]$. Aplicando a hipótese de indução e a regra de $\rightarrow$ELIM conclui-se que $\bar{\sigma} \sqsubseteq^* \bar{\sigma'} \Rightarrow A \cup \{x : \bar{\sigma}\} \vdash M_1 M_2 : \tau$. □

Lema 7.4 *Se $B \vdash M : \sigma$ e $\sigma \sqsubseteq^* \tau$ então $B \vdash M : \tau$.*

Demonstração: Vamos usar indução estrutural em M:

1. *Caso Base, $M \equiv x$*: Aplicação directa da regra $\sqsubseteq^*$.

2. *Passo de indução:*

(a) $M \equiv \lambda x.M'$: Pelo lema 7.2 $B \vdash \lambda x.M' : \bar{\sigma} \rightarrow \tau \Rightarrow B \cup \{x : \bar{\sigma'}\} \vdash M' : \tau$. Pela hipótese de indução para qualquer τ' tal que $\tau \sqsubseteq^* \tau'$ temos $B \cup \{x : \bar{\sigma'}\} \vdash M' : \tau'$. Pelo lema 7.3 para qualquer $\bar{\theta}$ tal que $\bar{\theta} \sqsubseteq^* \bar{\sigma'}$, $B \cup \{x : \bar{\theta}\} \vdash M' : \tau'$, logo para $\bar{\theta}$ e τ arbitrários tal que $\bar{\theta} \sqsubseteq^* \bar{\sigma'}$ e $\tau \sqsubseteq^* \tau'$ temos $B \cup \{x : \bar{\theta}\} \vdash M' : \tau' \Rightarrow B \vdash \lambda x.M' : \bar{\theta} \rightarrow \tau'$. Portanto, para todo o $\bar{\sigma'} \rightarrow \tau \sqsubseteq^* \bar{\theta} \rightarrow \tau'$ temos $B \vdash \lambda x.M' : \bar{\theta} \rightarrow \tau'$.

(b) $M \equiv M_1 M_2$: Pelo lema 7.2 $B \vdash M_1 M_2 : \sigma \Rightarrow \exists_{\bar{\theta}}[B \vdash M_1 : \bar{\theta} \rightarrow \sigma]$ e $\forall \theta_i \in \bar{\theta}[B \vdash M_2 : \theta_i]$. Pela hipótese de indução para toda a instância σ' de σ, como $\bar{\theta} \rightarrow \sigma'$ é uma instância de $\bar{\theta} \rightarrow \sigma$ temos $B \vdash M_1 : \bar{\theta} \rightarrow \sigma'$ e $\forall \theta_i \in \bar{\theta}[B \vdash M_2 : \theta_i]$. Portanto, pela regra $\rightarrow$ELIM temos $B \vdash M_1 M_2 : \sigma'$. $\square$

7.2.3 Expressividade

Em [Bak92], demonstrou-se que, para o sistema de Barendregt-Coppo-Dezani sem o tipo universal ω se, para um dado termo M, existem B e τ tal que $B \vdash M : \tau$, então, M é fortemente normalizável. Esse sistema engloba o sistema apresentado em 7.2, pois a regra de instância é mais geral, logo, para o sistema apresentado aqui temos $B \vdash M : \tau \Rightarrow M$ fortemente normalizável.

No mesmo artigo, é provado que se um termo é fortemente normalizável, então, admite tipo sem recurso à regra de instância. Usando o mesmo argumento, esse resultado aplica-se ao nosso sistema. Podemos, portanto, concluir que, com esta extensão ao sistema de Coppo-Dezani, mantemos a expressividade do sistema original, isto é, um termo admite tipo se e só se for fortemente normalizável.

7.2.4 Existência de Tipos Principais

A existência de tipos principais para o sistema de Coppo-Dezani tem permanecido um problema em aberto. O sistema mais próximo para o qual foi demonstrada a existência de tipos principais foi um sistema sem a constante ω e com a regra $\leq$ para termos do $\lambda-$calculus. Em [Ron88] definiu-se um procedimento que retorna o tipo principal de todos os termos fortemente normalizáveis. Como no sistema considerado, um termo admite tipo se e só se for fortemente normalizável, conclui-se que, se um termo admite um tipo, então, admite tipo principal.

O nosso sistema tipifica os mesmos termos que o sistema apresentado em [Ron88], os termos fortemente normalizáveis. Logo, para qualquer termo tipificável no nosso sistema temos um tipo principal. Basta aplicar o procedimento definido em [Ron88].

7.3 Restrições Induzidas por Intersecções

Vamos aqui definir um sistema de tipos com restrições, onde a linguagem de tipos é a linguagem de tipos com intersecções. Vamos seguir a notação habitual de $\bar{\sigma}$ para dizer que σ é uma intersecção de tipos.

Ao definir o sistema com restrições, limitamo-nos a transcrever as regras correspondentes do sistema com restrições inicial apresentado no capítulo 5 atendendo à maior expressividade da linguagem de tipos e a considerar como semântica das intersecções a intersecção das semânticas, conforme foi apresentado no capítulo 4. O sistema resultante é o seguinte:

$$AXIOMA \quad A * \{\bar{\sigma} \sqsubseteq \tau\} \cup R \vdash x : \tau, \text{ se } \bar{\sigma} \in A$$

$$\rightarrow ELIM \quad \frac{A * R \vdash M : \bar{\sigma} \rightarrow \tau', A * R_1 \vdash N : \tau_1, ..., A * R_n \vdash N : \tau_n}{A * R_1 \cup ... \cup R_n \cup R \cup \{\cap_i \tau_i \sqsubseteq \bar{\sigma}, \tau' \sqsubseteq \tau\} \vdash MN : \tau}$$

$$\rightarrow INTRO \quad \frac{A \cup \{x : \bar{\sigma_1}\} * R \vdash M : \tau'}{A * R \cup \{\bar{\sigma_2} \sqsubseteq \bar{\sigma_1}, \tau' \sqsubseteq \tau\} \vdash \lambda x.M : \bar{\sigma_2} \rightarrow \tau}$$

Vamos, agora, demonstrar a integridade do sistema:

Teorema 7.3 *Sejam A uma base, M um termo, σ um tipo e R um conjunto de restrições, tais que $A * R \vdash M : \sigma$, então $A * R \models M : \sigma$.*

Demonstração: Vamos usar indução estrutural em M.

1. Caso base: $M \equiv x$. Neste caso $A * \{\bar{\sigma} \sqsubseteq \tau\} \cup R \vdash x : \tau$. É trivial verificar que

$$\forall_\rho \forall_\varepsilon [\rho(x) \in T\|\bar{\sigma}\|\varepsilon \subseteq T\|\tau\|\varepsilon \Rightarrow \rho(x) \in T\|\tau\|\varepsilon]$$

   o que equivale a $A * \{\bar{\sigma} \sqsubseteq \tau\} \cup R \models x : \tau$.

2. $M \equiv M_1M_2$. $A * R_1 \cup ... \cup R_n \cup R \cup \{\cap_i \tau_i \sqsubseteq \bar{\sigma}, \beta \sqsubseteq \tau\} \vdash M : \tau$. Pela hipótese de indução, temos:

   (a) $A * R \models M_1 : \bar{\sigma} \rightarrow \beta$

   (b) $A * R_1 \models M_2 : \tau_1 \wedge ... \wedge A * R_n \models M_2 : \tau_n$

   Portanto, $\forall_\varepsilon \forall_\rho [\rho :_\varepsilon A \wedge R \Rightarrow E\|M_1\|\rho \in T\|\bar{\sigma} \rightarrow \beta\|\varepsilon]$ e $\forall_\varepsilon \forall_\rho [\rho :_\varepsilon A \wedge R_i \Rightarrow E\|M_2\|\rho \in T\|\tau_i\|\varepsilon \forall_{1 \leq i \leq n}]$. A primeira implicação garante-nos que para qualquer ρ tal que $\rho : A$ se verifica, $E\|M_1\|\rho$ é uma função que, quando aplicada a um elemento de $T\|\bar{\sigma}\|\varepsilon$, retorna um elemento de $T\|\beta\|\varepsilon$. Vamos assumir que $\forall_\varepsilon T\| \cap_i \tau_i\|\varepsilon \subseteq T\|\bar{\sigma}\|\varepsilon$. Dado que M_2 está simultâneamente em $T\|\tau_1\|\varepsilon$ e ... e $T\|\tau_n\|\varepsilon$ está em $T\| \cap_i \tau_i\|\varepsilon \forall_{1 \leq i \leq n}$, logo $\forall_\rho \forall_\varepsilon [R_1 \cup ... \cup R_n \cup R \cup \{\cap_i \tau_i \sqsubseteq \bar{\sigma}\} \wedge \rho :_\varepsilon A \Rightarrow (E\|M_1\|\rho)E\|M_2\|\rho \in T\|\beta\||]$. Se assumirmos $\beta \sqsubseteq \tau$ temos

   $$\forall_\varepsilon \forall_\rho [R_1 \cup ... \cup R_n \cup R \cup \{\cap_i \tau_i \sqsubseteq \bar{\sigma}, \tau' \sqsubseteq \tau\} \wedge \rho :_\varepsilon A \Rightarrow (E\|M_1\|\rho)E\|M_2\|\rho \in T\|\tau\|\varepsilon]$$

   que é equivalente a $A * R_1 \cup ... \cup R_n \cup R \cup \{\cap_i \tau_i \sqsubseteq \bar{\sigma}, \beta \sqsubseteq \tau\} \models M_1M_2 : \tau$.

3. $A * R \cup \{\bar{\sigma_2} \sqsubseteq \bar{\sigma_1}, \beta' \sqsubseteq \beta\} \vdash \lambda x.M : \bar{\sigma_2} \rightarrow \beta$. Queremos demonstrar que $\forall_\rho \forall_\varepsilon [\rho :_\varepsilon A \wedge R \cup \{\bar{\sigma_2} \sqsubseteq \bar{\sigma_1}, \beta' \sqsubseteq \beta\} \Rightarrow E\|\lambda x.M\|\rho \in T\|\bar{\sigma_2} \rightarrow \beta\|\varepsilon]$. Para isso basta demonstrar que $\forall_\rho \forall_\varepsilon [\rho :_\varepsilon A \wedge R \cup \{\bar{\sigma_2} \sqsubseteq \bar{\sigma_1}, \beta' \sqsubseteq \beta\} \Rightarrow E\|M\|\rho[v/x]$ está em $T\|\beta\|\varepsilon$ para todo o v em $T\|\bar{\sigma_2}\|\varepsilon]$.

   Pela hipótese de indução sabemos que $A \cup \{x : \bar{\sigma_1}\} * R \models M : \beta'$, ou seja $\forall_\varepsilon \forall_\rho [\rho :_\varepsilon A \cup \{x : \bar{\sigma_1}\} \wedge R \Rightarrow E\|M\|\rho \in T\|\beta'\|\varepsilon]$, logo para todo o v em $T\|\bar{\sigma_1}\|\varepsilon$, $E\|M\|\rho[v/x]$ está em $T\|\beta'\|\varepsilon$. Mas tendo como hipótese $\bar{\sigma_2} \sqsubseteq \bar{\sigma_1}$ o mesmo é verdade para $\bar{\sigma_2}$, ou seja, para todo o v em $T\|\bar{\sigma_2}\|\varepsilon$, $E\|M\|\rho[v/x]$ está em $T\|\beta'\|\varepsilon$. Como, por hipótese, $\beta' \sqsubseteq \beta$ o mesmo se passa para $T\|\beta\|\varepsilon$, o que demonstra o pretendido. □

7.4 Tipos Próprios

Um tipo com intersecções é um tipo polimórfico que representa o conjunto de tipos que podemos obter a partir dele por expansão, elevação e substituição. A definição da relação $\sqsubseteq_\cap$, como indispensável à derivação de qualquer dos tipos representados por um tipo com intersecções a partir dele, surge assim, naturalmente, tal como no polimorfismo paramétrico:

Definição 7.2 *Dado um tipo com intersecções σ e o conjunto T_σ de tipos representado por σ definimos a relação $\sqsubseteq_\cap$ como: $\sigma \sqsubseteq_\cap \tau$ se e só se $\tau \in T_\sigma$.*

Tal como acontecia no polimorfismo paramétrico, definimos agora um sistema de tipos que corresponde às soluções de conjuntos de restrições onde substituimos $\sqsubseteq$ por $\sqsubseteq_\cap$ quando a relação é aplicada a tipos polimórficos e por $=$ quando aplicada a tipos simples.

O sistema resultante, a que chamamos sistema próprio, é o seguinte:

$$AXIOMA \quad A * \{\bar{\sigma} \sqsubseteq_\cap \tau\} \cup R \vdash_\cap x : \tau, \text{ se } \bar{\sigma} \in A$$

$$\rightarrow ELIM \quad \frac{A * R \vdash_\cap M : \tau_1 \cap ... \cap \tau_n \rightarrow \tau, A * R_1 \vdash_\cap N : \tau_1, ..., A * R_n \vdash_\cap N : \tau_n}{A * R \cup R_1 \cup ... \cup R_n \vdash MN : \tau}$$

$$\rightarrow INTRO \quad \frac{A \cup \{x : \bar{\sigma_1}\} * R \vdash_\cap M : \tau}{A * R \cup \{\bar{\sigma_2} \sqsubseteq_\cap \bar{\sigma_1}\} \vdash_\cap \lambda x.M : \bar{\sigma_2} \rightarrow \tau}$$

Vamos ver algumas propriedades importantes deste sistema:

Lema 7.5 *No sistema próprio com intersecções $A \cup \{x : \bar{\sigma}\} \vdash_\cap M : \tau \wedge \bar{\sigma'} \sqsubseteq \bar{\sigma} \Rightarrow A \cup \{x : \bar{\sigma'}\} \vdash M : \tau$.*

Demonstração: Vamos usar indução estrutural em M.

Se x não ocorre em M a demonstração é trivial, pois o tipo de M não depende do tipo de x.

A demonstração segue para o caso em que x ocorre em M:

Caso base: $M \equiv x$. Neste caso $\{x : \bar{\sigma}\} * \{\bar{\sigma} \sqsubseteq \tau\} \vdash x : \tau$. Mas $\{x : \bar{\sigma'}\} * \{\bar{\sigma} \sqsubseteq \tau\} \vdash x : \tau$ pois $\bar{\sigma'} \sqsubseteq \bar{\sigma} \sqsubseteq \tau$.

Passos de indução: Os casos em que $M \equiv M_1 M_2$ e $\lambda y.N$ são triviais por hipótese de indução dado que as hipóteses em x se mantém após a aplicação das respectivas regras de derivação de tipos. □

Teorema 7.4 $A \vdash_\cap \lambda x.M : \bar{\sigma} \rightarrow \tau \Rightarrow A \vdash \lambda x.M : \bar{\sigma'} \rightarrow \tau, \forall_{\bar{\sigma}}.\bar{\sigma'} \sqsubseteq \bar{\sigma}$

Demonstração: $A \vdash_\cap \lambda x.M : \bar{\sigma} \to \tau \Rightarrow A \cup \{x : \bar{\sigma}\} \vdash_\cap M : \tau$. Pelo lema 7.5 temos $A \cup \{x : \bar{\sigma'}\} \vdash_\cap M : \tau \Rightarrow A \vdash_\cap \lambda x.M : \bar{\sigma'} \to \tau$. $\square$

O último teorema torna redundante a inequação da regra $\to INTRO$ no sistema próprio. O sistema próprio pode, assim, ser reformulado do seguinte modo:

$AXIOMA \quad A * \{\bar{\sigma} \sqsubseteq_\cap \tau\} \cup R \vdash_\cap x : \tau$, se $\bar{\sigma} \in A$

$$\to ELIM \quad \frac{A * R \vdash_\cap M : \tau_1 \cap ... \cap \tau_n \to \tau, A * R_1 \vdash_\cap N : \tau_1, ..., A * R_n \vdash_\cap N : \tau_n}{A * R \cup R_1 \cup ... \cup R_n \vdash MN : \tau}$$

$$\to INTRO \quad \frac{A \cup \{x : \bar{\sigma}\} * R \vdash_\cap M : \tau}{A * R \vdash_\cap \lambda x.M : \bar{\sigma} \to \tau}$$

O problema da falta de integridade da noção de instância tem, agora, uma nova leitura e justificação.

No sistema de Coppo-Dezanni, não é possível derivar o tipo $\alpha \to \beta \to (\alpha \cap \alpha_1 \to \beta)$ para o termo $\lambda x.x$, e no entanto, esse tipo é próprio com respeito ao mesmo termo, como é demonstrado pela seguinte derivação no sistema próprio com intersecções:

$$\begin{array}{c} \{x : \alpha \to \beta\} \vdash_\cap x : (\alpha \cap \alpha_1) \to \beta \\ \downarrow \\ \vdash_\cap \lambda x.x : (\alpha \to \beta) \to ((\alpha \cap \alpha_1) \to \beta) \end{array}$$

O problema surge no sistema de Coppo-Dezanni, porque esse sistema não deriva alguns tipos próprios. O sistema de Coppo-Dezanni parte de uma linguagem de tipos polimórficos que permite tipos como representações de conjuntos de tipos e não permite chegar, a partir de um dado tipo polimorfico, a todos os tipos que ele representa. Isto fica claro usando sistemas de tipos com restrições para definir quais os tipos próprios com respeito a um dado termo.

7.4.1 Relação com o Novo Sistema de Tipos com Intersecções

Tal como acontecia em relação aos sistemas de Curry e de Damas-Milner, este sistema corresponde ao novo sistema de tipos com intersecções definido em 7.2.2, isto é, o novo sistema de tipos com intersecções deriva, para cada termo, os tipos próprios com respeito a esse termo. Esse resultado está expresso no seguinte teorema:

Teorema 7.5 *Seja A uma base, M uma expressão, σ um tipo e R um conjunto de restrições. $A * R \vdash_\cap M : \sigma$ tem como significado que derivamos σ para M partindo de A no sistema próprio com intersecções e $A \vdash M : \sigma$ significa que derivamos σ para M a partir de A no novo sistema de tipos com intersecções definido em 7.2.2. Se o conjunto de restrições R se verificar, então:*

$$A \vdash M : \sigma \Leftrightarrow A \vdash_\cap M : \sigma$$

Demonstração: Vamos separar a demonstração do teorema em duas partes: uma que demonstra $A \vdash_\cap e : \sigma \Rightarrow A \vdash e : \sigma$ e outra $A \vdash e : \sigma \Rightarrow A \vdash_\cap e : \sigma$. Em qualquer dos casos, vamos usar indução estrutural em e. Só vamos apresentar a demonstração para o caso base em que e é uma variável x. Nos outros casos, as regras de derivação usadas são exactamente iguais pelo que se torna trivial verificar a equivalência dos sistemas.

Caso 1: $A \vdash x : \tau \Rightarrow A \vdash_\cap x : \tau$. Neste caso, temos $A \vdash x : \sigma_i$ onde $\sigma_i \in \bar{\sigma}$ e $\{x : \bar{\sigma}\} \in A$. Se $\sigma_i \in \bar{\sigma}$ então pela definição de elevação, $\bar{\sigma} \sqsubseteq_\cap \sigma_i$. Logo $A \vdash_\cap x : \sigma_i$.

O outro modo de derivar um tipo para x é pela regra $\sqsubseteq^*$ e nesse caso pela regra $AXIOMA$ do sistema próprio temos também $A \vdash_\cap x : \tau$.

Caso 2: $A \vdash_\cap x : \tau \Rightarrow A \vdash x : \tau$. Neste caso, temos $A \vdash_\cap x : \tau \Rightarrow \{x : \bar{\sigma}\} \in A \wedge \bar{\sigma} \sqsubseteq_\cap \tau$. Pela regra do AXIOMA em $\vdash$ temos $A \vdash x : \sigma_i \wedge \{x : \bar{\sigma}\} \in A \wedge \bar{\sigma} \sqsubseteq^* \sigma_i$. Mas pelo lema 7.1 sabemos que $\sigma_i \sqsubseteq^* \tau$, logo, pela regra $\sqsubseteq^*$ temos $A \vdash x : \tau$ como queríamos demonstrar. $\square$

7.5 Discussão

Neste capítulo, fomos motivados inicialmente pela origem dos sistemas com intersecções como um modo de permitir polimorfismo sem recorrer a uma regra de instância explícita. Embora a ideia de intersecções como polimorfismo seja bastante intuitiva, o sistema resultante obrigou à introdução de operações muito complexas, como a expansão, e apresentou, além disso, o problema de não ter uma noção de instância íntegra. Nós resolvemos esse problema acrescentado uma regra de instância somente nas variáveis.

Aplicámos, em seguida, a modelação de sistema de tipos por conjuntos de restrições desenvolvida nos dois capítulos anteriores a estes sistemas e ficou claro, sob o ponto de vista semântico, qual a falha do sistema original de Coppo-Dezanni: não podemos têr polimorfismo sem considerar soluções de desigualdades estritas no sistema com restrições que modela as intersecções. Esse resultado demonstrou que não podemos ter tipos com intersecções com boas propriedades sem uma regra explícita de instância.

Os resultados deste capítulo mostraram também a aplicação de restrições a sistemas de tipos onde os tipos são diferentes dos tipos simples e dos esquemas de tipos que modelam o polimorfismo paramétrico.

Outros sistema e outros conjuntos de tipos poderiam ser modelados por restrições. Pensamos que, a partir de qualquer semântica precisa para os tipos, chegamos facilmente ao sistema com restrições equivalentes. Fica, assim, em aberto um estudo mais lato de sistemas de tipos usando esta nova metodologia. Há, no entanto, duas extensões que queremos sugerir e que caem já fora do âmbito desta tese:

1. A aplicação de metodologias baseadas em restrições semânticas a outros sistemas de tipos com intersecções, nomeadamente, sistema que usem o tipo universal ω;
2. A procura de conjuntos de soluções das restrições associadas a cada tipo com intersecções para as quais existe um algoritmo de resolução. Isto levaria a sistemas de tipos que poderiam usar intersecções para modelar sobrecarga e que fossem úteis na construção de linguagens de programação.

Capítulo 8

Tipos em Programação em Lógica

8.1 Introdução

Tal como no λ-calculus, temos, no paradigma da programação em lógica, duas visões distintas de tipos: tipos prescritivos e descritivos. Na primeira categoria, temos sistemas que deram origem a linguagens de programação em lógica tipificadas como o Goedel, [HT92], ou o Typed Prolog, [LR91]. Na segunda categoria, temos sistemas baseados em aproximações da semântica do programa, como a interpretação abstracta, [BJ88] e [BG92], ou a determinação de modelos de um conjunto de restrições que são, elas próprias, aproximações do programa, [HJ90].

O nosso estudo assenta numa visão descritiva de tipos, portanto, a partir de agora, os tipos aparecem sempre com o sentido de propriedades de programas.

Neste contexto, como vamos ver em mais detalhe durante este capítulo, existem vários modos de estabelecer as disciplinas de tipos. Ao contrário do λ-calculus, em que a disciplina de tipos é sempre definida através de um conjunto de regras, em programação em lógica, essa definição pode assumir várias formas. Por exemplo, Mishra, [Mis84], usa árvores regulares para denotar a linguagem de tipos e um conjunto de regras de derivação semelhantes em estrutura, às regras clássicas usadas no λ-calculus. A partir de um programa e desse conjunto de regras, obtemos um conjunto de inequações nos tipos. Se esse conjunto tiver solução, o programa está bem tipificado, caso contrário, não. Para este sistema, programas mal tipificados não sucedem para nenhum golo. Note-se o contraste com o critério típico da programação funcional onde programas bem tipados não "falham".

Outra abordagem parte de uma linguagem de tipos que define um domínio abstracto num sistema de interpretação abstracta e vê tipos como aproximações do programa. Estas aproximações são conservativas no sentido em que, se um dado golo sucede, então, os argumentos são do tipo inferido previamente. Esta propriedade é garantida à custa da expressividade da linguagem de tipos (muitas vezes recorre-se à inferência de tipos gerais como o topo do reticulado que constitui o domínio abstracto para garantir a propriedade anterior). Um exemplo de um sistema deste género deve-se a Bruynooghe e Janssens, [BJ88] e usa uma representação de tipos através de uma classe de grafos correspondente a linguagens definidas por árvores regulares.

Para Zobel, [DZ92], tipos são regras de reescrita e é apresentado um algoritmo de inferência dessas regras baseado na identificação inicial de tipos dos argumentos dos programas e num algoritmo de simplificação do conjunto de tipos inicial.

Heintze, [HJ90], apresentou um sistema em que usa restrições num domínio de conjuntos para denotar tipos e um algoritmo de resolução dessas restrições para determinar se existe ou não uma solução (um tipo) para o programa. Finalmente, alguns autores, [FSVY91], usam programas em lógica com determinadas características para denotar tipos, os chamados programas regulares, e a inferência é feita, usando transformações do programa inicial para obter uma primeira versão dos programas de tipos e avaliação parcial para simplificar o conjunto de tipos obtido. Neste sistema, os tipos também são aproximações conservativas do programa, no sentido em que a semântica do programa está contida na semântica do programa de tipos.

Este capítulo apresenta uma descrição breve do paradigma da programação em lógica. Em seguida, apresenta os sistemas de tipos mais relevantes neste paradigma. Escolhemos estes sistemas, pois todos os outros se reduzem a qualquer um dos apresentados.

8.2 Programação em Lógica

Este capítulo tem como objectivo apresentar, de uma forma resumida, os conceitos necessários à apresentação dos resultados obtidos. Uma abordagem detalhada da teoria subjacente à área da Programação em Lógica pode ser encontrada em [Llo87].

8.2.1 Lógica

A lógica pode ser vista como a disciplina que estuda os padrões abstractos que estão na base do raciocínio. Usando como exemplo o raciocínio matemático, repare-se que a demonstração de um teorema não depende dos símbolos usados (podemos substituir os símbolos matemáticos por frases em português que a demonstração é equivalente), mas sim da estrutura e das regras utilizadas para relacionar estes símbolos.

Se conseguirmos abstrair os símbolos usados e focar a nossa atenção no modo como relacionamos essas entidades abstractas, conseguimos realçar o que realmente faz com que um raciocínio seja ou não válido.

Desde muito cedo que o homem tenta descrever, de um modo sistemático, a base do raciocínio. A primeira tentativa remonta à antiga Grécia e foi formulada por Aristóteles; para ele, um raciocínio consistia em afirmações, resultando cada uma delas das afirmações anteriores, através da aplicação de determinadas regras.

Desde então, baseados nesta ideia, vários sistemas aparecem com o mesmo objectivo. Nós estamos particularmente interessados na lógica de predicados em que os símbolos básicos da linguagem são propriedades, ou predicados.

8.2.2 Lógica de Predicados de Primeira Ordem

Aqui apresentamos uma introdução muito breve à lógica de primeira ordem. Partindo de um alfabeto de símbolos constituido por constantes, símbolos funcionais, variáveis e predicados, definimos o objecto básico desta lógica que é o *termo*.

Definição 8.1 *Um termo é uma constante, uma variável ou um símbolo funcional aplicado a um tuplo de termos. Um termo básico é um termo que não contém variáveis.*

Para definirmos o conjunto de fórmulas desta linguagem, consideramos mais duas classes de símbolos no alfabeto inicial: as conectivas, $\neg$, $\wedge$, $\vee$ $\rightarrow$ e $\leftrightarrow$, e os quantificadores $\forall$ e $\exists$.

Definição 8.2 *Uma fórmula bem formada é definida por:*

1. *Se p é um símbolo de predicado e $t_1, ..., t_n$ são termos então $p(t_1, ..., t_n)$ é uma fórmula designada por átomo;*

2. *Se F e W são fórmulas, então $(\neg F)$, $(F \wedge W)$, $(F \vee W)$, $(F \rightarrow G)$ e $(F \leftrightarrow G)$ são fórmulas;*

3. *Se F é uma fórmula e X uma variável, então, $(\forall_X F)$ e $(\exists_X F)$ são fórmulas.*

Definição 8.3 *Uma fórmula que não contém variáveis é uma fórmula básica.*

Se acrescentarmos a esta linguagem um conjunto de fórmulas que são sempre verdadeiras (os axiomas) e um conjunto de regras que permitem inferir fórmulas verdadeiras de fórmulas verdadeiras (as regras de inferência), temos uma teoria de primeira ordem. Nesta teoria, podemos denotar frases como *Para todo o X, se X é homem, então, X é mortal* como $\forall_X(homem(X) \rightarrow mortal(X))$.

8.2.3 Substituições

Nesta secção, referimos as definições de substituição e instância.

Definição 8.4 *Uma substituição θ é um conjunto finito da forma $\{X_1/t_1, ..., X_n/t_n\}$, onde $X_1, ..., X_n$ são variáveis diferentes e cada t_i é um termo diferente de X_i.*

Definição 8.5 *A aplicação de uma substituição $\theta = \{X_1/t_1, ..., X_n/t_n\}$ a uma fórmula F é a substituição, para $1 \leq i \leq n$, de toda a ocorrência de X_i em F por t_i. O resultado, representado por $F\theta$, é uma instância de F. Se $F\theta$ for uma fórmula básica, então, diz-se uma instância básica.*

8.2.4 Programação em Lógica

Em 1972, Robert Kowalski provou que a lógica podia ser usada como linguagem de programação. Isto deve-se a uma leitura procedimental de um tipo de fórmulas de lógica de primeira ordem, as cláusulas definidas. Uma cláusula definida é uma fórmula da forma $H \leftarrow B_1, ..., B_n$. Esta fórmula é lida do seguinte modo: H é verdade, se $B_1, ..., B_n$ forem, simultaneamente, verdade. A leitura procedimental proposta por Kowalsky é a de que H é um procedimento que consiste na execução dos procedimentos $B_1, ..., B_n$ (que por sua vez são definidos por outras cláusulas definidas). Além desta leitura, o que torna a lógica clausal uma linguagem de programação é a existência de uma estratégia de avaliação dos procedimentos - a estratégia de controle. Exemplos dessa estratégia são a procura depth-first no PROLOG ou outra qualquer regra de computação, existentes já em vários sistemas de execução paralela de programas em lógica.

Uma das principais virtudes deste paradigma é a de separar a componente descritiva do problema - as cláusulas propriamente ditas, do carácter operacional da linguagem - a estratégia de controle. Isto permite ao programador concentrar-se na descrição do problema em si, sem se preocupar com os métodos que a máquina vai utilizar para os resolver.

8.2.5 Sintaxe

Um programa em lógica é um conjunto de fórmulas com determinadas características - as cláusulas definidas. Antes desta definição, seguem algumas definições auxiliares.

Definição 8.6 *Um literal é um átomo ou a negação de um átomo. Um literal positivo é um átomo. Um literal negativo é a negação de um átomo.*

Definição 8.7 *Uma cláusula é uma fórmula da forma*

$$\forall_{X_1 ... X_n}(L_1 \vee ... \vee L_m)$$

em que L_i, $1 \leq i \leq n$ *é um literal e* $X_1 ... X_n$ *são as variáveis que ocorrem em* $L_1 \vee ... \vee L_m$.

Definição 8.8 *Uma cláusula definida contém exactamente um literal positivo e zero ou mais literais negativos.*

Uma cláusula definida é da forma $\forall_{X_1 ... X_n}(A \vee \neg B_1 \vee ... \vee \neg B_m)$ e surge normalmente na forma $A \leftarrow B_1, ..., B_n$. Assume-se que todas as variáveis da cláusula estão quantificadas universalmente e que o símbolo "," equivale à conjunção.

Definição 8.9 *Uma cláusula negativa, ou golo, não contém literais positivos e contém zero ou mais literais negativos.*

Um golo é da forma $\leftarrow B_1, ..., B_n$. Cada B_i é um subgolo.

Definição 8.10 *Uma cláusula negativa que não contém nenhum literal é uma cláusula vazia e representa-se por* $\Box$.

Definição 8.11 *Uma cláusula de Horn é uma cláusula definida ou um golo.*

Definição 8.12 *Um programa em lógica definido é um conjunto finito de cláusulas definidas.*

De um modo muito resumido, vamos descrever o processo de execução de um programa em lógica. Dado um golo $\leftarrow G_1, ..., G_n$, que pode ser visto como uma chamada de um procedimento, um passo da computação de $\leftarrow G_1, ..., G_n$ consiste na unificação de G_i com a cabeça de uma cláusula do programa, $A \leftarrow B_1, ..., B_m$ e a redução do golo $\leftarrow G_1, ..., G_n$ a $\leftarrow (G_1, ..., G_{i-1}, B_1, ..., B_m, G_{i+1}, ..., G_n)\theta$, onde θ é a substituição resultante da unificação de A com G_i. A computação termina com sucesso quando se chega ao golo vazio.

Exemplo 8.1 *Dado o programa:*

$$f(a).$$
$$p(X) \leftarrow f(X).$$

e o golo p(X), o resultado é a substituição $X = a$.

8.2.6 Semântica

Um programa em lógica é um conjunto de cláusulas definidas. Logo, a semântica declarativa de um programa é a semântica das fórmulas que o constituem.

8.2.6.1 Interpretações e Modelos

Como já foi referido, uma fórmula de lógica de primeira ordem é construida a partir de um conjunto de símbolos - o alfabeto. Os quantificadores e as conectivas têm um significado fixo: quando aplicados a fórmulas acerca das quais é conhecido o valor de verdade, retornam um determinado valor de verdade.

Exemplo 8.2 *$F_1 \wedge F_2$ é verdade se e só se F_1 é verdade e F_2 é verdade ou*

$(\forall_X)F$ é verdade se e só se para todo o valor de X, F é verdade.

O mesmo não acontece com os outros símbolos. As interpretações dão um significado aos restantes elementos da linguagem: constantes, functores, símbolos de variáveis e variáveis.

Definição 8.13 *Uma pre-interpretação consiste:*

1. *Num conjunto D, o domínio da interpretação. D é o domínio semântico dos símbolos da linguagem. Por exemplo, se usarmos lógica para exprimir aritmética, D é igual a N;*
2. *A cada constante da linguagem associamos um elemento de D;*
3. *A cada símbolo funcional de aridade n associamos uma função de D^n em D.*

Definição 8.14 *Uma interpretação I de uma linguagem L consiste numa pré-interpretação J com domínio D de L, e onde, a cada símbolo de predicado de aridade n de L é associada uma relação em D^n.*

Dizemos que $p(t_1, ..., t_n)$ é verdade se $< t_1, ..., t_n >$ pertence à relação associada a p pela interpretação I e é falso, caso contrário. Resumindo, um predicado define uma relação entre os seus argumentos.

Em programação em lógica, usamos um tipo específico de interpretações, as interpretações de Herbrand, onde cada símbolo se denota a si próprio. Não estamos interessados em interpretações que dão valores semânticos aos símbolos diferentes deles próprios, pois eles vão ser interpretados por um computador, para o qual só existem objectos sintácticos.

Partindo desta ideia, vamos definir, em primeiro lugar, o domínio semântico utilizado.

Definição 8.15 *O domínio de Herbrand de uma linguagem L é o conjunto de termos básicos que podemos construir usando as constantes e os símbolos de L.*

Exemplo 8.3 *Uma linguagem constituida pelo predicado p, pelos símbolos funcionais s e f e pela constante 0, tem como domínio de Herbrand o conjunto infinito* $\{0, s(0), f(0), s(s(0)), s(f(0)), f(s(0)), ...\}$.

Definição 8.16 *Dada uma linguagem L, a base de Herbrand é o conjunto de átomos básicos de L.*

Definição 8.17 *Uma interpretação de Herbrand é um subconjunto da base de Herbrand.*

A semântica é definida atribuindo o valor lógico *verdade* a todos os átomos que pertencem à interpretação e *falsidade* aos que não pertencem.

Definição 8.18 *Seja I uma interpretação de uma linguagem de primeira ordem L e F uma fórmula fechada em L. I é um modelo de F se F é verdadeira com respeito a I.*

Definição 8.19 *Seja S um conjunto de fórmulas fechadas de uma linguagem de primeira ordem L, e seja I uma interpretação de L. I é um modelo de S se I é um modelo de todas as fórmulas de S.*

Definição 8.20 *A instância básica de um programa P é o conjunto de todas as instâncias básicas de todas as suas cláusulas.*

Definição 8.21 *Dado um programa em lógica P, dizemos que uma interpretação de Herbrand I é um modelo de Herbrand de P se I é um modelo de todas as cláusulas da instância básica de P.*

Exemplo 8.4 *Consideremos a linguagem definida no exemplo 8.3 e o seguinte programa P:*

$$\begin{array}{l} int(0). \\ int(s(X)) \leftarrow int(X). \end{array}$$

P tem vários modelos. Por exemplo, a base de Herbrand da linguagem, o conjunto $\{0, f(0), s(0), s(s(0)), ...\}$ ou o conjunto $\{0, s(0), s(s(0)), ...\}$.

Dos vários modelos possíveis para um programa, escolhemos o modelo mínimo, isto é, a intersecção de todos os modelos, como a semântica de P. A escolha do modelo mínimo tem a ver com o facto de este ser o único que não contém átomos cujo valor de verdade não é consequência lógica do programa, logo, são verdade num modelo exterior a P, e, consequentemente, usando o programa, não podemos estabelecer a sua veracidade. Mais detalhes sobre este assunto podem ser encontrados em [EK76].

No decorrer deste trabalho, só vamos considerar programas definidos. Não consideramos casos em que podemos ter átomos negativos no corpo de uma cláusula. Quando isso acontece, a semântica de um programa é muito mais complexa, e preferimos considerar o caso mais simples dos programas definidos para evitar problemas que nos desviassem do objectivo do trabalho que é o estudo semântico de sistemas de tipos para programas em lógica.

8.2.6.2 O Operador T_P

Em [EK76] foi definido um operador que dada uma interpretação I_n e um programa P fornece uma nova interpretação I_{n+1} do seguinte modo:

$$T_P(I) = \{H | (H \leftarrow B_1, ..., B_n \text{ é uma instância básica de uma cláusula de } P \text{ e } \forall_{1 \leq i \leq n} B_i \in I\}$$

Emden e Kowalski demonstraram em [EK76] que o ponto fixo mínimo de T_P é o modelo mínimo de P. Toda a teoria subjacente a este resultado baseia-se em propriedades do conjunto de interpretações de um programa e em teoria do ponto fixo. A demonstração do resultado sai fora do âmbito deste trabalho, no entanto, remetemos o leitor para [EK76] ou qualquer livro de teoria da programação em lógica, por exemplo [Llo87].

Por um resultado conhecido da teoria do ponto fixo, o teorema de Kleene, (ver [Llo87]), sabemos que o ponto fixo mínimo de uma função contínua f definida num reticulado completo $[S, \leq]$, cujo elemento mínimo é $\perp$, é igual ao supremo do conjunto $\{f^k(\perp) | k \in N\}$. Quando $\leq$ é a inclusão de conjuntos $\subseteq$, o ponto fixo mínimo de f é igual a $\bigcup_{k \in N} f^k(\emptyset)$. O conjunto de todas as interpretações de Herbrand de um dado programa P é um reticulado completo sob a relação de ordem parcial inclusão de conjuntos, e com elemento mínimo $\emptyset$. Portanto, o supremo de $\{T_P^k(\emptyset)\}$ é o modelo mínimo de P. O supremo do conjunto anterior é representado por $T_P \uparrow \omega$.

Exemplo 8.5 *Dado o programa:*

$$p(a).$$
$$f(b) \leftarrow p(a).$$

temos

$$\begin{array}{rcl} T_P(\emptyset) & = & \{p(a)\} \\ T_P^2(\emptyset) & = & T_P(\{p(a)\}) \\ & = & \{f(b), p(a)\} \\ & = & T_P^3(\emptyset) \\ & = & modelo\ mínimo\ do\ programa \end{array}$$

Exemplo 8.6 *Dado o programa:*

$$\begin{array}{l} int(0). \\ int(s(X)) \leftarrow int(X). \end{array}$$

temos

$$\begin{array}{rcl} T_P(\emptyset) & = & \{int(0)\} \\ T_P^2(\emptyset) & = & \{int(0), int(s(0))\} \\ T_P^3(\emptyset) & = & \{int(0), int(s(0)), int(s(s(0))\} \\ & \vdots & \\ \bigcup_{k \in N} T_P^k(\emptyset) & = & \{int(0), int(s(0)), int(s(s(0)), ...\} \end{array}$$

Neste caso, o ponto fixo mínimo de T_P, e consequentemente, o modelo mínimo de P é um conjunto infinito.

Concluindo, a semântica de um programa em lógica P é dada pelo modelo mínimo de P. Um programa em lógica define um conjunto de relações em que cada uma delas é definida por um predicado.

8.3 Tipos em Programação em Lógica

Nesta secção, vamos fazer uma breve apresentação ao tema *Tipos em Programação em Lógica*. Como vamos ver, e ao contrário do λ-calculus, onde, apesar de uma grande diversidade de sistemas de tipos, eles são todos apresentados usando um formalismo comum, em Programação em Lógica, temos vários modos, aparentemente distintos, de apresentar teorias de tipos.

Tal como foi referido no ínicio da tese, este trabalho adopta uma visão descritiva de tipos. O mesmo se aplica em relação à Programação em Lógica, e, por isso, não vamos falar de sistemas prescritivos que deram origem a linguagens tipificadas como o Goedel, [HT92], o λ-Prolog, [NP92] e o Typed Prolog, [LR91].

Vamos apresentar quatro modos diferentes de apresentar sistemas de tipos descritivos para programas em lógica: árvores regulares, regras de reescrita, interpretação abstracta e programas em lógica regulares. Estas quatro abordagens cobrem, salvo diferenças pontuais, os sistemas de tipos descritivos conhecidos.

8.3.1 Árvores Regulares

Esta visão de tipos foi introduzida por Mishra, [Mis84]. Os tipos são descrições simbólicas de conjuntos de termos representadas por árvores regulares.

Definição 8.22 *Árvores regulares básicas (ARB) são definidas como:*

1. *O árvore vazia, $\emptyset$, é uma ARB;*
2. *Uma constante é uma ARB;*
3. *Se $a_1, ..., a_n$ são ARB então o tuplo $< a_1, ..., a_n >$ é uma ARB;*
4. *Dado um símbolo funcional f e uma ARB r a aplicação $f(r)$ é uma ARB;*
5. *Se t_1 e t_2 são ARB então $t_1 + t_2$ é uma ARB;*
6. *Se $g(x)$ é uma ARB onde ocorre a variável x, então fix $x.g(x)$ é uma ARB.*

8.3.1.1 Semântica

A semântica de uma árvore regular é o conjunto de termos que representa, e é dada por uma interpretação I definida do seguinte modo:

$$
\begin{array}{rcl}
I(\emptyset) & = & \{\} \\
I(c) & = & \{c\}, \text{onde } c \text{ é uma constante} \\
I(< a_1, ..., a_n >) & = & \{< t_1, ..., t_n > | t_i \in I(a_i)\} \\
I(f(r)) & = & \{f(t) | t \in I(r)\} \\
I(t_1 + t_2) & = & I(t_1) + I(t_2) \\
I(\mathit{fix}\ x.g(x)) & = & \bigcup_{i=0}^{\infty}\{g^i(\emptyset)\} = \text{menor conjunto de termos } X \text{ tal que} \\
 & & X = \{g(x) | x \in X\}
\end{array}
$$

8.3.1.2 Polimorfismo

Um tipo polimórfico é uma árvore regular que contém variáveis nos tipos. Tipos polimórficos denotam conjuntos de árvores regulares básicas. Sendo α uma variável que ocorre num tipo polimórfico $g(\alpha)$, a sua semântica é dada por:

$$\begin{aligned} I(g(\alpha)) &= \{I(g([\alpha/t]))|\text{onde } g([\alpha/t]) \text{ é a ARB resultante da substituição de } \alpha \\ &\quad \text{por qualquer ARB } t\} \end{aligned}$$

Exemplo 8.7 *Consideremos o seguinte programa em lógica:*

$$\begin{aligned} &append([], X, X). \\ &append([A|B], Y, [A|Z]) \leftarrow append(B, Y, Z). \end{aligned}$$

Os tipos para o primeiro, segundo e terceiro argumentos de append são, respectivamente:

1. *fix* $x.([] + [\alpha|x])$

2. β

3. *fix* $x.(\beta + [\alpha|x])$

8.3.1.3 Sistemas de Tipos

No λ-calculus, os sistemas de tipos são definidos à custa de uma relação $A \vdash M : \tau$. Neste caso, esta relação é condicional, no sentido em que temos $A \vdash M : \tau$ se S, onde S é um conjunto de restrições. O sistema de Mishra, [Mis84] é definido por:

$AXIOM \quad A \cup \vdash x : r$,se $r \in A$

$$CONS \qquad \frac{A \vdash t : r}{A \vdash f(t) : f(r)}$$

$$TUPLE \qquad \frac{A \vdash t_1 : r_1, \ldots, A \vdash t_n : r_n}{A \vdash < t_1, \ldots, t_n >:< r_1, \ldots, r_n >}$$

$$PRED \qquad \frac{\begin{array}{c} A \vdash t_1 : r_1, \ldots, A \vdash t_n : r_n, A \vdash b_1 : \text{ se } S_1, \ldots A \vdash b_n : \text{ se } S_n,, \\ p(t_1) \leftarrow b_1, \ldots p(t_n) \leftarrow b_n \end{array}}{A \vdash p : r_1 + \ldots + r_n \text{ se } S_1 \wedge \ldots \wedge S_n}$$

$$ATF \qquad \frac{A \vdash p : r, A \vdash t : s}{A \vdash p(t) : \text{ se } s \leq r}$$

$$CONJ \qquad \frac{A \vdash l_1 : \text{ se } S_1, A \vdash l_2 : \text{ se } S_2}{A \vdash l_1 \wedge l_2 : \text{ se } S_1 \wedge S_2}$$

$$TR \qquad \frac{A \vdash p : (r_1, \ldots r_n) + (s_1, \ldots, s_n)}{A \vdash p : (r_1 + s_1, \ldots, r_n + s_n)}$$

As restrições são introduzidas pela regra ATF (Fórmula atómica), que diz que uma fórmula atómica é bem tipificada se o tipo do argumento for um subtipo do tipo do predicado.

A regra TR define uma propriedade da operação +, a distributividade do tuplo, que faz com que esta linguagem de tipos seja determinística, num sentido que irá ser clarificado mais tarde. Nestes sistemas a noção de bem tipado reduz-se à resolução de um sistema de inequações e a correcção do sistema consiste em garantir que se um predicado não é tipificável, então não sucede para nenhum golo.

8.3.1.4 Inferência de Tipos Polimórficos

A correcção deste sistema é garantida para árvores regulares básicas. O mesmo não acontece quando existe polimorfismo, sendo este practicamente ignorado sob o ponto de vista de inferência.

8.3.2 Regras de Reescrita

Zobel em [Zob87] apresenta um sistema de tipos em que estes são definidos por meio de regras de reescrita. A linguagem de tipos é a mesma das árvores regulares, definindo o ponto fixo à custa de regras de reescrita recursivas. Neste sistema, um termo nos tipos é definido como:

1. Se α é uma variável de tipos, então, é um termo;
2. Se f é um functor e $\tau_1, \ldots, \tau_n$ são termos, então, $f(\tau_1, \ldots, \tau_n)$ é um termo.

Um tipo de um argumento de um predicado é dado por uma regra do género $\alpha \leftarrow S$, onde α é uma variável nos tipos e S um conjunto de termos nos tipos.

Exemplo 8.8 *O predicado*

$$\begin{array}{l} append([], X, X). \\ append([A|B], Y, [A|Z]) \leftarrow append(B, Y, Z). \end{array}$$

tem tipo $append(\eta_1, \eta_2, \eta_3)$ *onde:*

$$\begin{array}{l} \eta_1 \leftarrow \{[], [\alpha|\eta_1\} \\ \eta_3 \leftarrow \{\eta_2, [\alpha|\eta_3]\} \end{array}$$

Este sistema não é apresentado sob a forma habitual de um conjunto de regras de inferência que definem uma relação $M \vdash \tau$ com o significado M tem tipo τ. Em vez disso, o critério é semântico e é apresentado no trabalho em causa como um critério de correcção que garante que, se um predicado p não tiver tipo, então, nunca sucede. Isto significa que o conjunto de termos relacionados por p, quando aplicado aos tipos dos seus argumentos, contém o conjunto de termos pertencentes à relação definida por p.

A semântica do predicado está, assim, contida na semântica do tipo desse predicado. Como vamos ver, esse critério é usado na maior parte dos sistemas de tipos para programação em lógica como a única relação entre um objecto e o respectivo tipo. Isto levanta problemas como a decidibilidade do sistema ou a existência de determinadas propriedades dos sistemas de tipos, que não foram estudados até agora pela falta de uma definição mais precisa (mais próxima das definições de sistemas de tipos para o λ-calculus) da relação entre programas e os respectivos tipos.

Em [Zob87], a decidibilidade do sistema é apresentada à custa de um algoritmo que tipifica programas com tipos cuja semântica inclui a semântica do programa e que são "razoáveis"como tipos para esse programa. "Razoáveis", aqui, surge no sentido em que apesar de formalmente só garantirmos que incluem todos os golos que sucedem, também não são "gerais demais". Estas propriedades não são definidas e só se pode prever a sua existência pelo modo como o algoritmo de inferência é definido. No entanto, um estudo rigoroso deste género de sistemas deve começar por formalizar todas estas noções.

8.3.3 Interpretação Abstracta

A interpretação abstracta de programas foi introduzida, em [CC77], como uma técnica poderosa na análise automática de programas. Esta técnica consiste na definição de domínios abstractos cujos elementos são as propriedades que queremos extrair dos programas em tempo de compilação (propriedades estáticas), e em operações abstractas que correspondem à execução dos programas no domínio semântico abstracto.

Mais concretamente, dado um programa P, consideramos como domínio semântico de P, D_P, um reticulado completo com respeito a uma determinada relação de ordem $\leq_{D_P}$, e uma função E definida em D_P que representa a mudança de estado computacional. A semântica do programa é dada pelo ponto fixo de E, ou seja, pela solução da equação recursiva $x = E(x)$. Como E pode corresponder a uma computação infinita, esta semântica não pode ser usada para uma análise estática de P, entendendo propriedades estáticas de um programa como aquelas que podem ser derivadas sem ser necessário executar o programa. A solução é ter um domínio abstracto D_A (normalmente finito), onde é definida uma nova operação E_A que aproxima E.

Temos que, de algum modo, relacionar os domínios concretos e abstractos. Para isso definimos duas funções, α e γ, respectivamente a abstracção e a concretização, que permitem passar de D_P a D_A e de D_A a D_P. Sob certas condições, os objectos apresentados anteriormente, isto é, $< D_P, \leq_{D_P} >$, E, $< D_A, \leq_{D_A} >$, E_A, α e γ, formam uma interpretação abstracta de P. Essas condições são:

1. $< D_P, \leq_{D_P} >$ e $< D_A, \leq_{D_A} >$ são reticulados completos;

2. $E : D_P \rightarrow D_A$ e $E_A : D_A \rightarrow D_P$ são funções monótonas;

3. $\alpha : D_P \rightarrow D_A$ e $\gamma : D_A \rightarrow D_P$ são funções monótonas;

4. $\forall_{a \in D_A}, \alpha(\gamma(a)) = a$;

5. $\forall_{d \in D_P}, d \leq_{D_P} \gamma(\alpha(d))$

As condições 3, 4, e 5 constituem a definição de uma inserção de Galois entre $< D_P, \leq_{D_P} >$ e $< D_A, \leq_{D_A} >$. As inserções de Galois garantem que a aproximação do domínio concreto dada pelo domínio abstracto é íntegra, isto é, a concretização de um objecto a é, de certeza, um objecto que tem a propriedade a. Definida a interpretação abtracta, determinamos o ponto fixo de E_A e esta solução vai ser uma aproximação da solução da equação recursiva $x = E(x)$, que é a semântica de P. E_A é escolhido de modo a garantir que o ponto fixo de E_A é calculado após um número finito de passos.

Aplicando este método à inferência de tipos em programação em lógica, temos como domínio concreto de um programa P o reticulado de interpretações de P. A operação E pode ser o operador T_P (interpretação abstracta bottom-up, [BG92], [CDY90]) ou um passo de resolução (interpretação top-down, [Bru90]). Como domínio abstracto, temos um reticulado completo constituido pelos tipos. A função E_A é uma abstracção de E. O domínio abstracto é constituido por um reticulado de tipos e um tipo é uma árvore regular, [BJ88], que não é mais do que uma representação alternativa dos tipos regulares de Mishra.

Neste caso, as relações $\leq_{D_P}$ e $\leq_{D_A}$ denotam, repectivamente inclusão nos domínios concretos e abstractos. Logo, a condição 5) garante que a semântica de um dado objecto d está sempre contida na concretização da semântica do seu tipo. Como em programação em lógica, as semânticas de programas e tipos são ambas conjuntos de átomos, dizemos que a semântica de um tipo de um programa P contém a semântica de P.

8.3.4 Programas em Lógica Regulares

Alguns trabalhos sobre tipos em programação em lógica usam, como linguagem de especificação de tipos, uma classe de programas em lógica. Esses programas, denominados programas unários regulares [YS91], são definidos do seguinte modo:

Definição 8.23 *Um programa unário regular P é um programa em lógica com as seguintes propriedades:*

1. *Todos os predicados em P são unários;*
2. *Os argumentos das cabeças das cláusulas que definem um predicado não são unificáveis;*
3. *Os golos no corpo de todas as cláusulas de P são da forma $p(X)$, onde p é um símbolo de predicado e X é uma variável;*
4. *Toda a variável, numa cláusula, ocorre exactamente uma vez na cabeça e uma vez no corpo;*

Exemplo 8.9 *O seguinte programa é unário regular:*

$$
\begin{array}{l}
list([]). \\
list([X|R]) \leftarrow int(X), list(R). \\
int(0). \\
int(s(X)) \leftarrow int(X).
\end{array}
$$

Demonstra-se no mesmo artigo que estes programas definem exactamente os tipos regulares básicos de Mishra.

Um predicado p, de aridade n, de um programa P tem tipo T, onde T é um programa regular, se existem n predicados $t_1, \ldots, t_n \in T$ tais que:

1. $p^t(X_1, \ldots, X_n) \leftarrow t_1(X_1), \ldots, t_n(X_n)$;
2. Se $p(a_1, \ldots, a_n)$ pertence à semântica de p então $p^t(a_1, \ldots, a_n)$ pertence à semântica de p^t.

Existem duas abordagens distintas para a inferência de tipos. A primeira, [YS91], usa uma abstracção do operador T_P, que aproxima a semântica de P. A segunda, [Fru88], usa transformações do programa para chegar aos tipos. Esta última técnica consiste em considerar, primeiro, as projecções nos argumentos de cada predicado, o que fornece predicados unários. Em seguida, reduzem-se essas primeiras aproximações a uma forma normal.

8.3.4.1 Polimorfismo

Usando programas em lógica como tipos, o polimorfismo é conseguido à custa de predicados de ordem superior, [YFS92].

Exemplo 8.10 *O seguinte programa define o tipo lista:*

$$
\begin{array}{l}
list([]). \\
list([X|R]) \leftarrow \alpha(X), list(R).
\end{array}
$$

onde α é um predicado de ordem superior nos tipos. Isto permite susbtituir α por qualquer tipo básico e, consequentemente, define uma lista polimórfica.

Neste contexto, foram desenvolvidas linguagens de programação em lógica tipificadas polimorficamente, [YFS92].

8.4 Discussão

Como podemos ver, o conceito de disciplina de tipos em programação em lógica não está bem definido. Observa-se, no entanto, que existem propriedades semânticas comuns entre os vários sistemas existentes. Por exemplo, em todos eles, a semântica dos tipos é uma aproximação da semântica do programa.

Baseados nessa ideia, vamos ver, no próximo capítulo, que é possivel uma definição semântica geral da noção de programa bem tipificado. Tal como acontecia para o λ-calculus, também aqui vamos estudar algumas propriedades fundamentais dos sistemas de tipos, como a existência de tipos principais, usando essa metodologia mais geral.

Capítulo 9

Estudo Semântico de Sistemas de Tipos em Programação em Lógica

9.1 Introdução

Como vimos, as representações sintácticas de tipos descritivos para programas em lógica variam de sistema para sistema. Sob o ponto de vista semântico, no entanto, todas elas assentam na ideia comum de que τ é um tipo de um programa P se a semântica de P, estiver contida na semântica de τ.

A noção de programa tipificado, do género P tem tipo τ se ambos pertencerem a uma relação $\vdash$ entre programas e tipos, tal como no λ-calculus, não existe para a maior parte dos sistemas e surge em alguns sistemas, mas sempre definida à custa de operações externas, como sistemas de inequações ou algoritmos de unificação. Essas operações fazem sentido no processo de inferência de tipos e tornam as definições das relações $\vdash$ artificiais, quando comparadas com o que acontece para o λ-calculus. Isso não se deve a falhas na concepção dos sistemas de tipos descritivos até agora, mas sim a diferenças estruturais básicas entre os paradigmas funcional e lógico. Em programação em lógica, a definição da operação composição, trivial para funções e fundamental para a definição de tipos de objectos a partir dos tipos dos seus componentes, não está bem definida. Neste paradigma, o mais próximo desta operação para funções é a conjunção. De facto, tal como podemos definir novas funções à custa da composição, um predicado definido por uma cláusula de Horn está a ser definido à custa da conjunção de vários literais.

Uma tentativa de obter os tipos de uma cláusula, à custa dos tipos dos literais que nela estão envolvidos, levanta problemas bem conhecidos de quem trabalha em inferência de tipos para programação em lógica, como partilha de variáveis entre literais do corpo, definições recursivas que na "melhor das hipóteses" dão origem a inequações definidas pela noção de instância e, na pior das hipóteses, dão origem a ciclos infinitos ou a termos que crescem indefinidamente. No entanto, sem a definição de uma relação entre tipos e programas ou a identificação de condições necessárias à existência dessa relação, não podemos abordar problemas relevantes da teoria de tipos para programação em lógica, como a decidibilidade dos sistemas ou a existência de tipos principais.

As características próprias do paradigma da programação em lógica que fazem com que seja difícil e, permitimo-nos acrescentar, pouco natural, a definição de sistemas de tipos idênticos aos existentes para o λ-calculus, levou-nos a seguir uma abordagem semântica para obter uma caracterização semântica da relação entre programas e tipos.

Tal como no λ-calculus, existe, aqui, uma relação semântica $\models$ que define τ como tipo de P, se a semântica do segundo estiver contida na semântica do primeiro. Após o estudo apresentado na primeira parte desta tese de tipos para o λ-calculus e para o paradigma funcional, identificamos três propriedades básicas que relacionam tipos e programas:

- *Consistência:* se $A \vdash M : \tau$ e M se reduz a N por um passo básico de execução (no λ-calculus uma redução β) então $A \vdash N : \tau$;
- *Totalidade:* se $A \vdash M : \tau$, então, τ tipifica a semântica de M, $E\|M\|\rho$, para qualquer ρ, isto é, seja qual for o conjunto de valores básicos considerados para avaliar M;
- *Derivar tipos próprios:* como vimos, a definição de tipo próprio relaciona termos e tipos de um modo muito mais forte de que a integridade. Tal como no λ-calculus, também, em programação em lógica, não basta garantir a integridade dos sistemas, temos que garantir que, além de íntegros, os tipos sejam propriedades úteis dos programas.

Baseados nestas três propriedades, definimos propriedades semelhantes na programação em lógica. Aplicamos, em seguida, essas definições ao estudo de duas linguagens de tipos, os tipos regulares e os tipos regulares determinísticos, e chegamos à conclusão que, para a primeira, é impossível derivar tipos próprios e, na segunda, existem tipos próprios com respeito a todos os programas.

Todos os resultados apresentados são para programas em lógica definidos. Não lidamos com negação, embora pensemos que se trata de uma área onde é possível obtêr resultados interessantes.

Neste capítulo, começamos por apresentar a linguagem de tipos utilizada. Em seguida, relacionamos tipos e programas até chegar à noção de tipo próprio. Finalmente, aplicamos a metodologia desenvolvida ao estudo das linguagens de tipos regulares e determinísticas.

9.2 A Linguagem de Tipos

Como linguagem de especificação de tipos, escolhemos a programação em lógica. Esta escolha baseia-se, essencialmente, nas seguintes razões:

1. Tal como, na programação funcional, os tipos denotam o carácter funcional dos objectos, na programação em lógica, devem denotar o carácter relacional, logo, devem ser definidos por predicados e não por quaisquer outros objectos sintácticos a que, muitas vezes, por conveniência na definição de uma semântica, se dá uma leitura clausal;

2. Especificar tipos como programas em lógica fornece-lhe uma semântica natural: a semântica da programação em lógica;

3. A programação em lógica é uma linguagem de especificação suficientemente geral para representar os sistemas de tipos conhecidos para programação em lógica.

4. Tipos vão ser denotados por certas classes de programação em lógica, por exemplo, os programas unários regulares apresentados em [YS91]. Através do estudo de propriedades desses programas em lógica, podemos obter propriedades da linguagem de tipos.

Definição 9.1 *Um tipo regular é um programa unário, [FD92], da forma* $type_i(f(X_1, \ldots, X_n)) \leftarrow b_1 \wedge \ldots \wedge b_n$*, onde* $type_i$ *é um símbolo de predicado de tipos,* f *é um símbolo funcional,* X_i *são variáveis e* b_i *são literais da forma* $type_j(X)$ *onde* $type_j$ *é um símbolo de predicado de tipos e* X *é uma das variáveis* X_i*. Além disso, cada variável ocorre exactamente uma vez na cabeça e uma vez no corpo de cada cláusula.*

Os tipos são definidos à custa de predicados unários, pois cada um deles define uma aproximação de um conjunto de termos.

As restrições feitas a esses programas unários devem-se a razões de decidibilidade da linguagem de tipos.

O objectivo dos sistemas e das linguagens de tipos descritivos é fornecer descrições de propriedades dos programas, de modo a permitir detectar inconsistências ou efectuar optimizações em tempo de compilação. Dado que a semântica de um tipo é um conjunto de átomos básicos, é fundamental que o problema: *Dado um átomo p, p pertence ou não à semântica de um tipo T*, seja decidível. Usando vocabulário próprio da teoria da recursividade ([Phi92] e [Sho91]) temos que exigir que a semântica dos tipos sejam conjuntos recursivos. Só assim podemos saber, sem executar o programa, se um termo é ou não de um determinado tipo.

É também fundamental podermos comparar tipos. Recorda-se que a comparação de conjuntos recursivos não é decidível. Uma classe de linguagens que têm as propriedades da decidibilidade da comparação e intersecção é a das linguagens definidas por gramáticas de termos regulares. Um estudo completo desta classe de linguagens aplicada à especificação de tipos pode ser encontrado em [DZ92].

Os tipos definidos por gramáticas de termos regulares são definidos à custa de regras de reescrita iguais às definidas por Zobel e apresentadas em 8.3.2.

Teorema 9.1 *Os programas de tipos definidos por 9.1 definem exactamente os tipos definidos por gramáticas de termos regulares.*

Demonstração:

A cada regra de reescrita $\alpha \rightarrow S$, é associado um predicado α, tal que para cada tipo $\tau \in S$ temos uma cláusula $\alpha(t) \leftarrow \beta_1(X_1) \wedge ... \wedge \beta_k(X_k)$, onde $t = \tau$ com cada ocorrência de um símbolo de tipos β_i substituido por uma nova variável X_i. É trivial verificar que estas cláusulas pertencem à classe definida em 9.1. Inversamente, dada uma cláusula $h(t) \leftarrow b_1(X_1) \wedge ... \wedge b_k(X_k)$ ela é transformada na regra $h \rightarrow \tau$, onde τ é t com cada variável X_i substituida pelo tipo b_i. □

9.2.0.2 Tipos Determinísticos

Apesar da comparação e intersecção de tipos regulares ser decidível, dados dois tipos regulares T_1 and T_2, nem sempre é possível obter o seu unificador mais geral ([DZ92]).

Exemplo 9.1 *Consideremos os dois tipos:*

$$t_1(f(a,b)).$$
$$t_1(f(c,d)).$$

e

$$t_2(f(X,Y)) \leftarrow \alpha(X), \beta(Y).$$

α e β são variáveis de tipos. O unificador mais geral de t_1 e t_2, se existir, é da forma $t_2(f(X,Y)) \leftarrow \alpha(X), \beta(Y)$. Mas não existe nenhuma substituição de α e β que aplicada a t_2 tenha como resultado t_1, portanto, não existe unificador mais geral de t_1 e t_2.

Existe uma subclasse dos tipos regulares para a qual é decidível o problema de encontrar o unificador mais geral entre dois tipos. São os chamados tipos regulares determinísticos:

Definição 9.2 *Um tipo regular T é determinístico se, para cada predicado p, os argumentos de p em cláusulas diferentes têm functores principais diferentes e os argumentos das cabeças de todas as cláusulas não são variáveis.*

Os tipos determinísticos correspondem aos programas regulares definidos em [YS91].

9.2.0.3 Distributividade do Tuplo

Distributividade do tuplo, [Mis84], é uma propriedade dos conjuntos de termos que consiste na independência entre conjuntos de valores em duas posições diferentes nos argumentos do functor. Seguindo esta ideia inicial, dado um conjunto de termos S, podemos construir uma aproximação de S, S' com a propriedade da distributividade do tuplo; a menor aproximação nessas condições é o fecho distributivo de S. O fecho distributivo de um conjunto de termos S é uma aproximação de S onde o contra-domínio da operação substituição não é o conjunto de termos, mas sim o conjunto dos conjuntos de termos, isto é, o conjunto de tipos. A maior parte dos sistemas de inferência de tipos para programação em lógica usam este género de aproximação da semântica dos programas como a semântica dos tipos. A definição desta propriedade é a seguinte:

Definição 9.3 ([YFS92]) *Seja S um conjunto de termos básicos. O fecho distributivo de S, $\alpha(S)$, é definido do seguinte modo:*

$$\alpha(S) = \{c \mid c \text{ é uma constante} \wedge c \in S\} \cup \{f(t_1, ..., t_n) \mid t_i \in \alpha(f_{n,i}^{-1}(S)), 1 \leq i \leq n\}$$

onde f é um símbolo funcional em S e $f_{n,i}^{-1}(S)$ é definido como:

$$f_{n,i}^{-1}(S) = \{t_i \mid f(t_1, ..., ti, ..., t_n) \in S\}.$$

Um conjunto S é distributivo no tuplo se $S = \alpha(S)$.

Esta definição é facilmente extensível a conjuntos de átomos unários:

Definição 9.4 *Seja S um conjunto de átomos unários. S é distributivo no tuplo se, para cada predicado, o conjunto de termos definido por esse predicado for distributivo no tuplo.*

Demonstramos que a classe de tipos definidos por programas de tipos determinísticos é a mesma classe que é definida por conjuntos de termos distributivos no tuplo:

Teorema 9.2 *O modelo mínimo de um programa de tipos deterministico é um conjunto distributivo no tuplo.*

Demonstração:

A demonstração usa indução no número de iteracções do operador T_P, quando P é um programa de tipos deterministico.

Caso base: seja P um programa de tipos determinístico. O conjunto de factos de P, $T_P(\emptyset)$ é constituido por factos da forma $t(t_i)$ onde para cada predicado t, os argumentos t_i não têm o mesmo functor principal. Portanto, $T_P(\emptyset)$ é distributivo no tuplo.

Passo de indução: temos que demonstrar que o conjunto de termos S_{k+1} definido por predicados de $T_P^{k+1}(\emptyset)$ é tal que $S_{k+1} = \alpha(S_{k+1})$. A hipótese de indução é que o conjunto de termos S_k, definido por predicados de $T_P^k(\emptyset)$, é distributivo no tuplo, isto é, $S_k = \alpha(S_k)$.

Se o termo $t \in S_{k+1}$ for uma constante, $t \in S_{k+1}$ e $t \in \alpha(S_{k+1})$ trivialmente.

No caso de $t \in S_{k+1}$ e $t \notin S_k$ e t da forma $f(t_1, \ldots, t_n)$:

1. $f(t_1, \ldots, t_n) \in S_{k+1} \Rightarrow f(t_1, \ldots, t_n) \in \alpha(S_{k+1})$, pois o segundo conjunto é uma aproximação conservativa do primeiro;

2. $f(t_1, \ldots, t_n) \in \alpha(S_{k+1}) \Rightarrow \forall_{1 \leq j \leq n}(t_j \in \{t_i | f(t_1, \ldots, t_i, \ldots, t_n) \in S_{k+1}\})$. Mas, pela definição do operador T_P e, atendendo a que

$$p(f(t_1, \ldots, t_i, \ldots, t_n)) : -p_1(t_1), \ldots, p_i(t_i), \ldots, p_n(t_n)$$

   , $t_i \in S_k$.

Pelas mesmas razões, e atendendo a que a última cláusula é a única a definir termos da forma $f(t_1, \ldots, t_n)$, pois P é determinístico, concluimos que $f(t_1, \ldots, t_i, \ldots, t_n) \in S_{k+1}\})$, como queriamos demonstrar. Para os elementos de S_{k+1} que pertencem a S_k o resultado verifica-se trivialmente aplicando a hipótese de indução. □

9.2.1 Polimorfismo Paramétrico

Como já foi referido, o polimorfismo paramétrico é conseguido usando predicados de ordem superior. Vamos agora definir as noções de cláusula de tipos paramétrica e esquema de tipos. Chamamos parâmetros a esses predicados de ordem superior.

Definição 9.5 *Uma cláusula de tipos paramétrica é uma cláusula de tipos onde todos os átomos $p(X)$ são tais que p é um símbolo de predicado ou de parâmetro.*

Definição 9.6 *Um esquema de tipos T é um conjunto finito de cláusulas de tipos paramétricas.*

Definição 9.7 *Dado um programa em lógica P, um esquema de tipos de P consiste num esquema de tipos que para todo o predicado p de aridade n, define os predicados de tipo $type_i^p$ para $i = 1...n$.*

Exemplo 9.2 *Seja P a definição de append:*

$$append([], X, X).$$
$$append([X|R], Y, [X|R1]) \leftarrow append(R, Y, R1).$$

Um esquema de tipos para append é:

$$
\begin{aligned}
& type_1([]). \\
& type_1([X|R]) \leftarrow \alpha(X), type_1(R). \\
& type_2(X) \leftarrow \beta(X). \\
& type_3(X) \leftarrow \beta(X). \\
& type_3([X|R]) \leftarrow \alpha(X) \wedge type_3(R).
\end{aligned}
$$

Definição 9.8 *Dado um programa de tipos T, os parâmentros $\alpha_1, ..., \alpha_n$ e os predicados de tipos $\tau_1, ..., \tau_n$, $[\tau_1/\alpha_1, ..., \tau_n/\alpha_n]$ define uma substituição genérica dos parâmetros α_i por τ_i e dos outros parâmetros pelo predicado geral $\alpha(X)$.*

Definição 9.9 *Um programa de tipos T_1 é uma instância de um esquema de tipos T_2 se e só se existe uma susbstituição genérica C tal que $T_1 = CT_2$.*

Tal como acontece no λ-calculus, um esquema de tipos denota um conjunto de programas de tipo, um para cada instância genérica de cada parâmetro.

9.2.2 Semântica

Vamos agora relacionar a semântica dos programas com os respectivos tipos. Vamos supor que um programa P é tipificado por um programa de tipos T. A semântica do programa tipificado não é constituida pelo conjunto de átomos que validam P, mas sim pelo conjunto de átomos que validam P e cujos argumentos são consequências lógicas de T.

Dizemos que um termo t é uma consequência lógica de um esquema de tipos T, $T \vdash t$, se existe uma instância genérica de T, T', tal que t pode ser derivado de T' usando um método de inferência. Note-se que os programas de tipos formam uma classe decidível das cláusulas de Horn, logo $T' \vdash t$ é computável em tempo finito.

Definimos a noção de modelo de um programa P tipificado por T do seguinte modo:

Definição 9.10 *Dado um tipo T e um programa em lógica P, uma interpretação I é um modelo bem tipificado de P, se valida todas as cláusulas de P e se os argumentos de todos os elementos de I são consequências lógicas de T.*

Definição 9.11 *A semântica de um programa em lógica P tipificado por T, $SEM_T(P)$, é o modelo mínimo bem tipificado de P.*

9.2.3 Tipos e Programas

Nos sistemas de tipos para o λ-calculus existe uma relação bem definida entre expressões e tipos: $\vdash E : \tau$ se τ pode ser obtido a partir de um conjunto de regras de derivação de tipos. Como vimos em capítulos anteriores, existem condições semânticas necessárias para um tipo τ tipificar uma expressão E. Para o λ-calculus, essa condição resume-se a τ ser próprio com respeito a M.

Neste estudo, vamos estabelecer condições semânticas necessárias à relação entre programas e tipos.

9.2.3.1 Tipos Consistentes

Dado um programa em lógica P, tipificado por um programa de tipos T, temos que garantir que, dado um golo G que sucede em P, T tipifica todos os subgolos necessários à demonstração de G. Esta condição garante que um tipo é consistente com todas as possíveis computações efectuadas por um programa. Tipos consistentes são equivalentes, no lambda calculus, à propriedade de manter os tipos de expressões por reduçao β, isto é, se $\vdash M : \tau$ e M reduz-se, por redução β a N, então $\vdash N : \tau$.

Esta propriedade também garante que não precisamos de fazer inferência de tipos em tempo de execução, pois um tipo de um programa tipifica todas as possíveis execuções do programa.

Definição 9.12 *Dado um programa em lógica P e um tipo T, T é consistente com P se e só se para todos os golos G que sucedem em P, se T valida os argumentos de G, então, T valida os argumentos de todos os nós da árvore de derivação de G.*

Exemplo 9.3 *O tipo:*

$$
\begin{array}{l}
t([]).\\
t([\_]).\\
t([\_,\_]).\\
t([\_,\_,\_]).
\end{array}
$$

é consistente com o programa que define uma lista. O tipo que consiste no facto $t([\_,\_,\_])$ *não é consistente com o mesmo programa, pois para demonstrar que* $[\_,\_,\_]$ *é uma lista, temos que provar que* $[\_,\_]$, $[\_]$ *e* $[]$ *também são listas.*

9.2.3.2 Tipos Completos

Outra classe de tipos é a classe dos tipos completos.

Definição 9.13 *Dado um programa em lógica* P *e um programa de tipos* T, T *é completo se e só se, para toda a instância básica de cada cláusula de* P*, se* T *valida todos os argumentos de todos os predicados no corpo, então,* T *também valida os argumentos da cabeça.*

Esta condição garante que a teoria de tipos respeita uma semântica completa do programa, por exemplo, a semântica obtida à custa do operador T_P. O tipo apresentado no exemplo 9.3 é consistente, mas não é completo com respeito a um programa que define listas.

9.2.3.3 Tipos Próprios

Tal como acontece no λ-calculus, os tipos consistentes e completos são, muitas vezes, gerais demais com respeito a um determinado programa.

Exemplo 9.4 *Consideremos o programa constituido pelo facto:*

$$p(f(a)).$$

e os três programas de tipos:

T_1 :	T_2 :	T_3 :
$t(f(a))$.	$t(f(a))$.	$t(f(a))$.
	$t(f(X))$.	$t(X)$.

T_2 *e* T_3 *são redundantes se atendermos ao facto de que queremos definir* $\{p(f(a))\}$*.*

Queremos evitar que, dado um programa P, o tipo de P não contenha cláusulas redundantes. No caso anterior, esse tipo seria T_1.

Vamos, para isso, definir uma relação de ordem parcial nos tipos, e considerar tipos mínimos de entre aqueles que alteram a semântica do programa do mesmo modo.

Essa relação de ordem é definida à custa da noção de intersecção de programas de tipos.

Definição 9.14 (adaptada de [BMPT90]) *Dados dois tipos T_1 e T_2, $T_1 \cap T_2$ é o tipo obtido do seguinte modo:*

Se $p(t) \leftarrow Body_1 \in T_1$ e $p(u) \leftarrow Body_2 \in T_1$ e $\exists \theta = mgu(t, u)$ então $p(t\theta) \leftarrow (Body_1, Body_2)\theta$ é uma cláusula de $T_1 \cap T_2$.

Definição 9.15 *Dados dois tipos T_1 e T_2, $T_1 \sqsubseteq T_2$ se e só se $T_1 \cap T_2 = T_1$.*

Definição 9.16 *Dado um tipo T consistente e completo com respeito a um programa P, T é próprio com respeito a P se e só se $\forall_{T'}$ se $SEM_T(P) = SEM_{T'}(P)$ então $T \sqsubseteq T'$.*

Teorema 9.3 *Dado um programa em lógica P e um programa de tipos T, T é próprio com respeito a P se e só se:*

$$T = \bigcap_{SEM_T(P) = SEM_{T'}(P)} T'$$

Demonstração:

O resultado é trivial atendendo à definição de tipo próprio como um tipo tal que $\forall_{T'}$ se $SEM_T(P) = SEM_{T'}(P)$, então $T \sqsubseteq T'$, e à definição da relação $\sqsubseteq$ como $T_1 \sqsubseteq T_2$ se e só se $T_1 \cap T_2 = T_1$. $\Box$

Vamos, agora, apresentar algumas aplicações da definição de tipo próprios ao estudo de sistemas de tipos para a programação em lógica.

Esta propriedade permite-nos definir a propriedade do tipo principal sem recorrer ao sistema de tipos, ou seja, de um modo puramente semântico. A definição de tipo principal é:

Definição 9.17 *Um esquema de tipos T_0 é um tipo principal de um programa P se e só se:*

1. *T_0 é próprio com respeito a P;*
2. *Qualquer tipo próprio com respeito a P é uma instância de T_0.*

A existência de tipos próprios é, deste modo, uma condição semântica necessária à existência de tipos principais. Essa condição permite-nos, à partida, investigar a existência ou não de tipos principais, sem definir o sistema de tipos explicitamente. Os dois próximos exemplos mostram duas dessas aplicações.

Vamos investigar a existência de tipos próprios, definidos à custa da intersecção de todos os tipos que modificam a semântica do programa do mesmo modo, para as linguagens regulares e regulares deterministicas.

Consideremos o seguinte exemplo:

Exemplo 9.5 *Consideremos o seguinte conjunto infinito de tipos regulares:*

T_0 :
$t([0]).$
$t([X|R]) \leftarrow int(X), t(R).$

T_1:
$t([0]).$
$t([s(0), 0]).$
$t([X_0, X_1, X_2|R]) \leftarrow int(X_0), int(X_1), int(X_2), t(R).$

T_i:
$t([0]).$
$\vdots$
$t([s^i(0), ..., 0])$
$t([X_0, \ldots, X_i, X_{i+1}|R]) \leftarrow int(X_0), ..., int(X_i), int(X_{i+1}), t(R).$

$int(0).$
$int(s(X)) \leftarrow int(X).$

Estes programas são regulares e a sua intersecção:

$t([0]).$

$$t([s(X), X|R]) \leftarrow t([X|R]).$$

não é regular.

Este exemplo demonstra o seguinte:

Teorema 9.4 *Na classe de tipos regulares, nem sempre é possível definir tipos próprios com respeito a um dado programa.*

Como corolário óbvio, temos a não existência de tipos regulares principais.

Isto demonstra que , tal como no λ-calculus, uma maior expressividade do sistema de tipos, aqui somente definido pela linguagem de tipos utilizada, sacrifica propriedades que o tornam praticável para uma análise estática de programas, como a propriedade do tipo principal.

Essa propriedade verifica-se na classe de tipos determinísticos.

Antes de demonstrar esse resultado, vamos apresentar algumas propriedades das linguagens regulares determinísticas.

Lema 9.1 *Sejam S_i, $1 \leq i \leq \infty$ um número infinito de conjuntos distributivos no tuplo e*

$$S = \bigcap_{1 \leq i \leq \infty} S_i$$

então S é distributivo no tuplo.

Demonstração:

Seguimos uma técnica de redução ao absurdo. Vamos supor que existem $f(t_1, \ldots, t_n)$ e $f(s_1, \ldots, s_n) \in S$, tal que, para uma posição i, $f(t_1, \ldots, s_i, \ldots t_n) \notin S$. Isto quer dizer que $f(t_1, \ldots, t_n)$ e $f(s_1, \ldots, s_n)$ pertencem a todos os S_i e existe um S_i que não contém $f(t_1, \ldots, s_i, \ldots t_n)$, para um i arbitrário. Logo, esse S_i não é distributivo no tuplo, o que contraria a hipótese. Portanto, S é distributivo no tuplo. $\square$

Lema 9.2 *Dado um conjunto distributivo no tuplo, S, cujos elementos são construidos a partir de um número finito de símbolos, existe um programa regular determinístico, que tem S como modelo mínimo.*

Demonstração:

Cada predicado p definido em S pode ser descrito do seguinte modo: dado que os elementos de S são construidos à custa de um alfabeto finito, para cada p existe um número finito de functores $f_i, 1 \leq i \leq n$, tal que, para cada f_i, temos p definido como:

1. $p(f)$, se f tem aridade 0 ou
2. $p(f(X_1, ..., X_m))$ se $X_1 \in S_1 \wedge ... \wedge X_m \in S_m$, onde $S_1, ...S_m$ são conjuntos de termos distributivos no tuplo e construidos a partir de símbolos que ocorrem em S.

Se associarmos a cada S_i um predicado unário p_i que define os termos pertencentes a S_i, temos:

1. $p(f(X_1, ..., X_m)) \leftarrow p_1(X_1), ..., p_n(X_n)$. (com algum p_i eventualmente igual a p) ou:
2. $p(f)$.

Para cada p_i, aplicamos o o mesmo raciocínio usado para p, pois define um conjunto distributivo no tuplo, e temos k_i cláusulas que o definem. k_i é finito para todo o i, pois S usa um número finito de símbolos, logo no final deste processo, temos um número finito de cláusulas da forma 1. ou 2., que constituem um programa regular determinístico. $\Box$

Teorema 9.5 *Dado um conjunto, que pode ser infinito, de programas regulares determinísticos, a sua intersecção é um programa regular determinístico.*

Demosntração: Cada programa regular determinístico tem uma semântica que é um conjunto determinístico no tuplo. Pelos teoremas anteriores, a intersecção de todas as semânticas ainda é determinístico no tuplo, e é o modelo mínimo de um programa regular deterministico. $\Box$

Teorema 9.6 *Dado um programa em lógica P, existem, na classe de tipos regulares determinísticos, tipos próprios com respeito a P.*

Demonstração: Pelo teorema 9.5, a intersecção de quaisquer tipos regulares determinísticos é um tipo regular determinístico. O resultado é implicado pelo teorema 9.3. □

A garantia da existência de tipos regulares determinísticos próprios, juntamente com a existência de unificadores mais gerais para esta classe de tipos, leva-nos a considerar como muito provável a existência de tipos principais nesta classe. Isto faz dos tipos regulares determinísticos excelentes candidatos à construção de bons sistemas de tipos para programas em lógica.

9.3 Discussão

Neste capítulo, definimos para o paradigma da programação em lógica critérios semânticos que relacionam tipos com programas. Esse critérios eram dados, ou à custa de algoritmos de inferência, o que os torna impraticáveis quando os sistemas não são decidíveis, ou à custa da integridade do sistema. Este último critério, embora necessário, não é suficiente para relacionar tipos e programas de modo a que os primeiros sejam descrições úteis dos segundos.

Depois de chegar à noção de tipo próprio com respeito a um determinado programa como condição semântica à tipificação do programa pelo tipo, vimos que, só a partir de uma análise da linguagem de tipos utilizada, podemos verificar se é possível derivar tipos próprios.

Pensamos ser possível ir mais longe do que as linguagens determinísticas para denotar tipos em programação em lógica. O problema da definição de boas linguagens de tipos passa por áreas diferentes como a teoria da recursividade ou a terminação de programas e é, neste momento, uma área prometedora de investigação.

É importante referir que a utilização de programação em lógica para denotar tipos em programação em lógica não é nova. (ver [FSVY91]). A nossa utilização dessa metodologia realça a facilidade da mesma para:

1. Definir propriedades das linguagens de tipos;

2. Relacionar, sob o ponto de vista semântico, tipos e programas.

Finalmente, queremos deixar algumas indicações para um trabalho futuro na mesma área:

1. Relação entre os nosso critérios semânticos para relacionar tipos e termos e os critérios matemáticos usados na interpretação abstracta;
2. Aplicar os nossos critérios à classificação de domínios de interpretação abstracta;
3. Investigação na área de classes recursivas de programação em lógica como possíveis linguagens de tipos;

Capítulo 10

Conclusões

10.1 Conclusões Gerais

O nosso programa de investigação foi o estudo e levantamento de problemas em sistemas de tipos descritivos para linguagens declarativas, com vista à obtenção de uma caracterização semântica desses mesmos sistemas. O estudo foi feito para o λ-calculus e para a programação em lógica. Chegou o momento de reflectir sobre os resultados atingidos.

10.1.1 λ-calculus

Nos sistemas de tipos descritivos para o λ-calculus, a única caracterização semântica dos mesmos sistemas é uma aproximação semântica e a exigência de integridade em relação a essa aproximação. A relação $\models$ define o conjunto de tipos admissíveis sob o ponto de vista semântico e os sistemas de tipos são definidos usando outra relação, $\vdash$, a que se exige integridade em relação a $\models$. Neste contexto, a definição de sistemas de tipos com restrições como sistemas gerais de derivação de tipos para o λ-calculus e a identificação de subconjuntos de soluções das restrições como os tipos que os sistemas devem derivar para garantir certas propriedades fornece, pela primeira vez, uma caracterização semântica dos sistemas de tipos. Essa caracterização exige o respeito de outras propriedades além da integridade, e que achamos fundamentais e justificadas pelo objectivo pretendido, quando derivamos descrições de termos. Além disso, conseguimos uma metodologia geral que vê tipos como um conjunto de restrições e exige aos sistemas de tipos integridade com respeito a um certo modo de resolver essas restrições. Quando essa integridade não existe, os sistemas podem apresentar comportamentos patológicos, como vimos, quando estudámos os sistemas de tipos com intersecções. Desde que a integridade forte seja garantida, o próprio sistema pode ser visto como um resolutor de restrições. A nossa metodologia abre caminho a trabalho futuro na área de inferência de tipos como resolução de restrições, uma área de investigação que pensamos ser bastante promissora.

10.1.2 Programação em Lógica

No paradigma da programação em lógica, a integridade é não só a única propriedade semântica exigida aos sistemas de tipos descritivos como, muitas vezes, subjacente à própria definição dos sistemas. Neste paradigma, a maior parte dos sistemas baseia-se numa aproximação conservativa da semântica do programa. Além disso, as notações e métodos usados para a derivação de tipos diferem consideravelmente de sistema para sistema, o que dificultou a nossa tarefa. Conseguimos chegar a uma definição de tipo próprio que relaciona programas com tipos, mas não aplicámos essa relação ao estudo de sistemas existentes neste paradigma, pelo que o objectivo final não foi totalmente atingido. No entanto, a metodologia desenvolvida demonstrou ser suficientemente geral para permitir um estudo comparativo de linguagens de tipos. Com a aplicação da nossa noção de tipos próprios, concluímos que os tipos regulares não são adequados neste paradigma e confirmamos a adequação dos tipos regulares determinísticos.

10.2 Caracterização Semântica de Sistemas de Tipos

Uma primeira abordagem para a caracterização semântica do sistema de Curry sugeria escolher, de entre todos os tipos válidos sob o ponto de vista semântico, aqueles que fornecessem mais informação, ou seja, os tipos mínimos com respeito a uma relação de inclusão semântica. Por exemplo, partindo do principio que $\|int\| \subseteq \|real\|$, temos $\models \lambda x.x : int \rightarrow real$ e $\models \lambda x.x : int \rightarrow int$. Neste caso, escolheríamos $int \rightarrow int$ como sendo próprio com respeito a $\lambda x.x$ pois $\|int \rightarrow int\| \subseteq \|\sigma\|$ para todo o σ comparável com $int \rightarrow int$ tal que $\models \lambda x.x : \sigma$. O problema com esta definição é quando num termo $\lambda x.M$ x não ocorre livre em M. Neste caso, desde que τ seja próprio com respeito a M o tipo mínimo sob o ponto de vista semântico para $\lambda x.M$ seria $\top \rightarrow \tau$ em que $\top$ seria um elemento máximo do conjunto de tipos. Isto, porque temos que permitir $\alpha \to \tau$ como descrição do termo $\lambda x.M$ para qualquer α, pois $\lambda x.M$ não é função de x.

Este problema resolve-se com a escolha do tipo próprio, não como o tipo mínimo, mas como a solução mínima de um sistema de restrições induzida pela relação semântica $\models$. Só assim conseguimos capturar as relações entre tipos que, sob o ponto de vista semântico, são impostas pela própria estrutura do termo, o que foi a motivação inicial da definição de sistemas de tipos para o λ-calculus. A este respeito, concluímos que essas relações são respeitadas pelos sistemas de Curry e de Damas-Milner e que, nestes casos, a escolha de soluções minímas corresponde a um modo de resolver restrições, usando, respectivamente, a unificação e um caso especial da semi-unificação. É confortável verificar que os dois algoritmos usados para resolver restrições nestes dois sistemas coincidem com os respectivos algoritmos de inferência de tipos.

10.3 Tipos com Intersecções

Como vimos no decorrer da tese, os sistemas com intersecções surgem com o objectivo de expandir o polimorfismo, de modo a permitir uma enumeração finita dos tipos admissíveis como argumentos de funções. O sistema inicial modela polimorfismo, na medida em que permite derivar $x : \sigma_i$ de $x : \sigma_1 \cap ... \cap \sigma_n$ para todo o i entre 1 e n.

Um tipo polimórfico, neste caso, tem uma estrutura diferente dos vários tipos que representa. Isto levanta os seguintes problemas:

1. Não é possível usar a substituição para chegar de um tipo polimórfico a qualquer tipo representado por ele;

2. Perde-se o isomorfismo entre árvores de derivação de tipos para o mesmo termo.

O primeiro problema é resolvido definindo as operações de expansão e elevação que nos parecem pouco intuitivas como definição de instância. O problema anterior torna qualquer demonstração directa de propriedades dos sistema de tipos com intersecções que tenham a ver com a derivação de tipos, uma tarefa muito difícil. Além disso, o sistema de Coppo-Dezanni não tem em conta o facto dos tipos com intersecções serem tipos polimórficos, na medida em que não permite, a partir de um tipo polimórfico, chegar a todos os tipos por ele representado.

Mesmo que se consigam subconjuntos decidíveis destes sistemas, os problemas apresentados levam-nos a pôr em causa a sua aplicabilidade prática na definição de sistemas de tipos para linguagens de programação.

10.4 Tipos em Programação em Lógica

Neste paradigma, atente-se nos seguintes resultados existentes:

1. Definição de sistemas de tipos como no λ-calculus não foram aplicados na prática para a inferência de tipos descritivos;

2. Os métodos de inferência de tipos neste paradigma usados com algum sucesso na prática, são baseados em aproximações conservativas da semântica do programa: interpretação abstracta ou aproximação directa do próprio programa por outro programa cuja semântica é uma aproximação do primeiro.

Os resultados obtidos nesta tese apontam para:

1. Uma caracterização semântica da relação entre tipos e programas assente directamente na semântica e não, como acontece no λ-calculus, num sistema estruturalmente induzido por essa semântica;

2. A caracterização semântica obtida é aplicável ao estudo de linguagens de tipos, independentemente do sistema em causa.

Analisando os pontos anteriores, verificamos que há, realmente, uma diferença estrutural entre os dois paradigmas que torna mais natural para a programação em lógica a definição de tipos como aproximação directa da semântica dos programas. Essa aproximação deve obedecer a certas propriedades que, neste paradigma, se prendem mais a propriedades das linguagens usadas para denotar as aproximações do que propriemente ao modo de as obter a partir do programa.

Um dos nossos objectivos iniciais foi tentar ver até que ponto, a nível semântico, as relações entre termos e tipos são semelhantes nos paradigmas lógico e funcional. A única relação entre os tipos próprios, nos dois paradigmas, é que em ambos eles são mínimos sob o ponto de vista semântico, ou melhor, são as melhores descrições possíveis dos objectos que descrevem de entre todas as descrições comparáveis por um critério de inclusão semântica. Este resultado abre novos caminhos no tema das relações existentes entre tipos em programação funcional e em programação em lógica, atendendo sempre às diferenças estruturais entre os dois paradigmas e sem cair na tentação de transferir para um paradigma os resultados e metodologias existentes no outro.

10.5 Trabalho Futuro

Aqui pretendemos deixar algumas pistas para possíveis extensões ao trabalho apresentado nesta tese.

10.5.1 Inferência de Tipos como Resolução de Restrições

Os sistemas de tipos com restrições definidos nesta tese apresentam tipos como soluções de um conjunto de restrições. O modo de obter essas soluções define o sistema de tipos. No caso do sistema de Curry, recorremos ao algoritmo da unificação e, no caso do sistema de Damas-Milner, ao algoritmo de inferência de tipos da linguagem ML. Fica para trabalho futuro a tarefa de implementar um sistema de inferência do conjunto de restrições associadas a cada termo. Em seguida, recorreríamos a um resolutor de restrições. Este resolutor seria a própria definição do sistema de tipos.

As potencialidades desta área futura de trabalho aberta por resultados obtidos nesta tese assentam no seguinte:

1. Ser o cruzamento de duas áreas bastante férteis de investigação em ciência de computadores: a inferência de tipos e a programação por restrições;

2. Permitir, usando técnicas de resolução de restrições baseadas no atraso da resolução de restrições que não conseguimos resolver, tornar úteis para implementação de linguagens de programação, sistemas de tipos bastante expressivos mas não decidíveis;

3. Tornar os sistemas de inferência de tipos mais gerais, modulares e baseados directamente nas relações semânticas entre termos e tipos. Primeiro, inferíamos um conjunto de restrições que, posteriormente, resolveríamos usando a estratégia que melhor se adaptasse aos objectivos práticos da inferência em causa.

10.5.2 Estudo da Relação entre Restrições e Sistemas mais Gerais

Há extensões aos sistema de Damas-Milner que permitem ter recursividade polimórfica ,[Myc84], ou polimorfismo nas variáveis quantificadas por λ ,[Rey74]. Esses sistemas extremamente expressivos, pagam a expressividade com a falta de decidibilidade. Na nossa tese, a aplicação de sistemas com restrições ao estudo de sistemas de tipos ficou pelos sistemas com intersecções. Pensamos que um estudo similar para os sistemas referidos no início poderia explicar, sob o ponto de vista semântico, qual a razão de algumas propriedades dos sistemas em causa como, por exemplo, a falta de decidibilidade ou a ausência de tipos principais.

10.5.3 Sistemas com Intersecções

A única extensão do trabalho efectuado sobre sistemas com intersecções parece ser a definição, a partir de sistemas com a noção de instância íntegra, de subsistemas decidíveis que pudessem dar origem a modelos de sobrecarga em linguagens de programação. Nesta conclusão, permitimo-nos ter uma opinião subjectiva sobre este assunto: dado o modo pouco natural das definições de expansão e a propriedade deste tipo de polimorfismo que é perder o isomorfismo entre árvores de derivação de tipos para um mesmo termo, pomos sérias reservas quanto à aplicabilidade destes sistemas na construção de linguagens de programação.

10.5.4 Linguagens de Tipos para Programação em Lógica

Apesar de termos demonstrado a não adequação de linguagens regulares para denotar tipos em programação em lógica, pensamos que podemos ir mais longe em expressividade do que as linguagens determinísticas. Estamos, neste momento, a estudar características de uma nova linguagem de tipos mais expressiva do que as linguagens regulares determinísticas onde esperamos ser possível definir tipos próprios com respeito a qualquer programa. Essa nova linguagem baseia-se numa adaptação da caracterização de funções que define as funções primitivas recursivas, [Phi92]. Estamos a dar uma nova leitura a essa noção no contexto de relações e predicados primitivos recursivos. Este trabalho não está concluído, mas dá fortes indicações de que a área de investigação da definição de boas linguagens de tipos para programação em lógica está longe de estar esgotada.

10.5.5 Interpretação Abstracta

Existe uma clara identificação de linguagens de tipos com domínios de interpretação abstracta que é a seguinte: ao vermos um tipo como uma aproximação semântica de um programa, podemos ver a linguagem de tipos como a definição de um novo domínio semântico onde podemos definir novos programas (tipos). Posteriormente, podemos executar esses novos programas (tipos) para obter aproximações da execução do programa inicial. Esta é a ideia subjacente à interpretação abstracta. Pensamos que o estudo de linguagens de tipos efectuada nesta tese poderia ser adaptado ao estudo de domínios de interpretação abstracta. Seria útil relacionar os critérios usados na definição de tipos próprios com os critérios matemáticos (inserções de Galois) usados para definir interpretações abstractas. Esse trabalho permitiria ligar uma nova abordagem assente na lógica com uma abordagem assente em propriedades matemáticas dos domínios e funções de aproximação.

10.6 Síntese

Chegou o momento de verificar até que ponto cumprimos o objectivo inicial de obter uma caracterização semântica do conjunto de tipos vistos como descrições de um dado objecto.

Pensamos que, se compararmos o critério usual, que se resume à exigência de integridade, com o critério apresentado aqui, que exige que os sistemas derivem tipos próprios com respeito aos termos, podemos afirmar que contribuimos consideravelmente para essa caracterização. Podem-se discutir as escolhas dos critérios que estão na base da definição de tipos próprios. Nós tentámos sempre, ao definir estes tipos, respeitar somente as relações semânticas entre tipos e termos derivadas do facto dos primeiros serem descrições dos segundos, que é a ideia base dos sistemas de tipos descritivos.

Finalmente, queremos salientar que, ao definir os tipos próprios para o λ-calculus, iniciámos o que pode vir a ser uma nova área de investigação, tendo como base uma visão de tipos como sistemas de restrições e de sistemas de tipos como modos de resolver essas restrições.

Referências

[Bak92] S. van Bakel. Complete restrictions of the intersection type discipline. *Theoretical Computer Science*, 102:135–163, 1992.

[Bak93a] S. van Bakel. Essential intersection type assignement. In *13 conference on foundations of software technology and theoretical computer science*, volume 761 of *lecture notes on computer science*, pages 13–23. Springer-verlag, 1993.

[Bak93b] S. van Bakel. Principal type schemes for the strict type assignemet system. *Logic and Computation*, 3(6):643–6670, 1993.

[BCD83] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignement. *Journal of symbolic logic*, 48(4):931–940, 1983.

[BG92] R. Barbuti and R. Giacobazzi. A bottom-up polymorphic type inference in logic programming. *Science of Computer Programming*, 1992.

[BJ88] M. Bruynooghe and G. Janssens. An instance of abstract interpretation integrating type and mode inference. In *Proc. of the 5th International Conference and Symposium on Logic Programming*, 1988.

[BMPT90] A. Brogi, P. Mancarella, D. Pedresci, and F. Turini. Composition operator for logic theories. In *Proc. on the Symp. on Computational Logic*. Spriger-Verlag, 1990.

[Bru90] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. In *Journal of Logic Programming*, 1990.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conf. Rec. 4th Acm Symp. on Prin. of Programming Languages*, 1977.

[CD80] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Journal of Formal Logic*, 21(4):685–693, 1980.

[CDV80] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal type schemes and λ-calculus semantics. In J.R. Hindley and J.P. Seldin, editors, *To H.B. Curry, Essays in combinatory logic, lambda-calculus and formalism*, pages 535–560. Academic Press, 1980.

[CDV81] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift fur Mathematische Logik und Grundlagen der Mathematik*, 27:45–58, 1981.

[CDY90] M. Codish, D. Dams, and E. Yardeni. Bottom-up abstract interpretation of logic programs. Technical report, Weizmann Institute of Science, 1990.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 1940.

[Chu41] A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[Chu33] A. Church. A set of postulates for the foundation of logic. *Anals of Mathematics*, 2(33-34):346–366 and 839–864, 1932/33.

[Cop80] M. Coppo. An extended polymorphic type system for applicative languages. *Lecture Notes in Computer Science*, 88, 1980.

[Cur34] H.B. Curry. Functionality in combinatory logic. In *Proc. Nat. Acad. Sci. U.S.A.*, volume 20, pages 584–590, 1934.

[Dam84] Luis Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1984.

[DM82] L. Damas and R. Milner. Principal type schemes for functional languages. In *9th ACM Symposium on Principles of Programming Languages*, 1982.

[DZ92] P. Dart and J. Zobel. A regular type language for logic programs. In Frank Pfenning, editor, *Types in Logic Programming*. The MIT Press, 1992.

[EK76] M. Van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of ACM*, 23, 1976.

[EST95] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. *Proceedings of OOPSLA 95*, 1995.

[FD92] M. Florido and L. Damas. Types as theories. In *Proc. of post-conference workshop on Proofs and Types, Joint International Conference and Symposium on Logic Programming*, 1992.

[Flo91] M. Florido. A constraint solver for IC-prolog II. Master's thesis, Imperial College University of London, 1991.

[Fru88] T. Fruhwirth. Type inference by program transformation and partial evaluation. In *Proceedings of the Workshop on Meta-Programming in Logic Programming*, 1988.

[FSVY91] T. Fruhwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proc. 6th IEEE Symp. on Logic for Computer Science*, 1991.

[Gir72] J.Y. Girard. *Interpretation fonctionelle et élimination des coupures dans l'arithmetique d'ordre superieur*. PhD thesis, Universite de Paris, 1972.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2), April 1993.

[Hin69] J.R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[HJ90] N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Proc. of the ACM Symposium on Principles of Programming Languages*, 1990.

[HT92] P.M. Hill and R.W. Topor. A semantics for typed logic programs. In Frank Pfenning, editor, *Types in Logic Programming*. The MIT Press, 1992.

[JJ87] J.L. Lassez J. Jaffar. Constraint logic programming. In *Conf. Rec. 14th Acm Symp. on Prin. of Programming Languages*, 1987.

[KTU90] A. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. *22nd ACM Symposium on Theory of Computation*, 1990.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.

[LR91] T. Lakshman and U. Reddy. Typed prolog: a semantic recostruction of the Mycroft-O'Keefe type system. In *Proc. of the 1991 International Symposium on Logic Programming*, 1991.

[Mil40] R. Milner. A theory of type polymorphism in programming. *Journal of Symbolic Logic*, 1940.

[Mil84] R. Milner. A proposal for standard ML. In *ACM Symposium on LISP and Functional Programming*, 1984.

[Mis84] P. Mishra. Towards a theory of types in Prolog. In *Proc. of the 1984 International Symposium on Logic Programming*, 1984.

[Myc84] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Poc. 6th International Conference on Programming*, volume 167 of *Lecture Notes in Computer Science*, 1984.

[NP92] G. Nadathur and F. Pfenning. Types in higher order logic programming. In Frank Pfenning, editor, *Types in Logic Programming*. The MIT Press, 1992.

[Phi92] I. Phillips. Recursion theory. *Handbook of Logic in Computer Science*, 1, 1992.

[Plo76] G. Plotkin. A power domain construction. *SIAM Journal of Computation*, 3(5):452–487, 1976.

[PS96] J. Palsberg and S. Smith. Constrained types and their expressiveness. *ACM Transactions on Programming Languages*, 18(5):519–527, 1996.

[Rey74] J.C. Reynolds. Towards a theory of type structure. In Spinger-Verlag, editor, *Mathematical Foundations of Software Development*, volume 173 of *Lecture Notes in Computer Science*, pages 408–425, 1974.

[Rob65] J. A. Robinson. A machine oriented logic based on the resolution principle. *Journal of ACM*, 12(1):23–41, 1965.

[Ron88] S. Ronchi della Rocca. Principal type scheme and unification for intersection type discipline. *Theoretical Computer Science*, 59:181–209, 1988.

[RV84] S. Ronchi della Rocca and B. Venneri. Principal type schemes for an extended type theory. *Theoretical Computer Science*, 28:151–169, 1984.

[Sho91] J. R. Shoenfield. *Recursion Theory.* Springer-Verlag, 1991.

[Sto77] J. Stoy. *Denotational Semantics: The Scott-Strachey approach to Programming Languages.* MIT Press, 1977.

[Wel94] J. B. Wells. Typability and type-checking in the second-order λ-calculus are equivalent and undecidable. In *Logic in Computer Science*, pages 176–185, 1994.

[YFS92] E. Yardeni, T. Fruehwirth, and E. Shapiro. Polymorphic typed logic programs. In Frank Pfenning, editor, *Types in Logic Programming.* The MIT Press, 1992.

[YS91] E. Yardeni and E. Shapiro. A type system for logic programs. In *The Journal of Logic Programming*, 1991.

[Zob87] J. Zobel. Derivation of polymorphic types for prolog programs. In *Proceedings 4th International Conference on Logic Programming*, 1987.