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

Polimorfismo de registros no Sistema CT

João Rafael Moraes Nicola

2 de dezembro de 2006

Orientadores:
Carlos Camarão
Lucília Figueiredo

Resumo

Registros são usados em diversas linguagens (sob diversos nomes) para expressar a composição de objetos de dados e são usualmente associados a uma ou mais operações, como a projeção, atualização e remoção de campos. Tais operações são consideradas polimórficas se puderem ser usadas com qualquer registro que satisfaça um conjunto pequeno de exigências, e.g. uma operação de atualização de um determinado campo deve poder ser usada com qualquer campo que possua o campo em questão, independente do tipo do campo e da presença ou não de outros campos no registro. Diversas propostas foram feitas para o acréscimo de operações polimórficas de registro em Haskell. Este trabalho propõe uma abordagem baseada no sistema de restrições do Sistema CT e usa restrições para codificar os requerimentos de cada operação sobre registros. A proposta inclui alterações à sintaxe e ao sistema de tipos do Sistema CT. Para resolver o problema da satisfazibilidade das restrições criadas para codificar as exigências das operações, foi desenvolvida uma teoria que serviu de base para um algoritmo de satisfazibilidade. O trabalho também inclui uma sintaxe alternativa, que pode ser usada para apresentar de forma amigável os tipos inferidos para operações de registro.

"Àquele que está assentado
no trono e ao Cordeiro, seja o
louvor, e a honra, e a glória,
e o domínio pelos séculos dos
séculos."

Ap. 5:13

À Priscila e ao Daniel e seus irmãozinhos.

Agradecimentos

Este trabalho não poderia ter sido realizado, não fosse pela ajuda de muitas pessoas, que me apoiaram durante todo o período. A todos estes deixo minha gratidão:

Aos meus pais, pelo incentivo e suporte sempre presente e, principalmente, por acreditarem em meu potencial desde que, quando criança, sonhava com este mundo da ciência e dos computadores.

À minha esposa querida, Priscila, que sempre foi e sempre será, imprescindível em todos os meus desafios: obrigado pela alegria, paciência, disposição e carinho e, acima de tudo, pelo amor com que suportou tantos sacrifícios.

Ao meu filho, Daniel, que, mesmo sem se dar conta, foi para mim uma fonte inesgotável de alegria e ânimo: quem não se alegraria diante do seu sorriso?

Aos meus orientadores, Carlos e Lucília: obrigado pela amizade, pelo exemplo, pelos diálogos, pelos conselhos, pela paciência, pelas correções e por confiarem em minha capacidade, mesmo quando dela eu duvidava.

Aos meus irmãos, Pedro e André, pela amizade e pela paciência.

Aos meus professores: em especial ao prof. Newton, que me ensinou a construir argumentos formais, e ao prof. Bigonha, pelo estágio em docência, o qual permitu-me aprofundar meu conhecimento em Sistemas de Reescrita, e pelos comentários sobre este documento.

Ao prof. Alberto Pardo, pelo incentivo e por se dispor a vir de tão longe para participar da banca.

Aos meus colegas de curso, por me aguentarem, e em especial ao Kristian, que foi meu grande companheiro.

Aos meus queridos amigos de Belo Horizonte: obrigado pelo carinho, pela amizade e pelo amor. Obrigado ao Eugênio e Carol, ao Edílson e Regina e à Leca e Paulinho, que me receberam em suas casas como um irmão ou um filho.

Ao meu tio Carlos, que me deu o meu primeiro livro sobre computadores, e que me deu muitas diárias de hotel: muito obrigado, tio!

Ao meu tio Pipico, pelo incentivo.

Aos meus avós, por suportarem minha ausência.

Ao Vinicius e a Eline, obrigado pela paciência e amizade.

E a todos os amigos e parentes, que me incentivaram e aguardaram, com paciência, o término deste longo trabalho.

Sumário

Sumário **6**

Lista de Figuras **8**

1 Introdução **9**

2 Visão geral da linguagem Haskell **13**

3 Registros em Haskell **23**
3.1 Declarações de registros 23
3.2 Problemas com o projeto atual de registros em Haskell 28
3.3 Alternativas 31

4 Sistema CT **33**
4.1 Classes de tipos de Haskell 33
4.2 Sintaxe 35
4.3 Substituições 38
4.4 Visão geral do Sistema CT 39
4.5 Semi-reticulados 40
4.6 Operações de supremo 43
4.7 Satisfazibilidade de conjuntos de restrições 45
4.8 Regras de satisfazibilidade 47
4.9 Sistema de tipos 48
4.10 Inferência de tipos 49
4.11 Substituições principais 50

5 Acrescentando polimorfismo de registro ao Sistema CT **53**
5.1 Objetivos do projeto 53
5.2 Operações de registros 53
5.3 Sistema de tipos 58
5.4 Inferência de tipos 65

6 Conjuntos de campos **69**
6.1 Definições e lemas . . . 69
6.2 Operações sobre conjuntos de campos . . . 73
6.3 Termos de conjuntos de campos . . . 84
6.4 Substituições . . . 84
6.5 Equações de conjuntos de campos . . . 88
6.6 Sistemas de equações de conjuntos de campos . . . 92
6.7 A forma $\preceq$-normal . . . 93
6.8 Forma pré-normal . . . 100
6.9 O sistema de redução $\mathcal{W}$ . . . 101
6.10 Propriedades de terminação do sistema de redução . . . 109
6.11 De conjuntos de restrições a instâncias de problemas . . . 116
6.12 Exemplo de redução . . . 117

7 Sintaxe de usuário para polimorfismo de registro **119**
7.1 Sintaxe amigável . . . 119
7.2 Traduzindo da sintaxe de usuário para a sintaxe interna . . . 122
7.3 Traduzindo da sintaxe interna para a sintaxe de usuário . . . 124
7.4 Exemplo . . . 126

8 Implementação **129**
8.1 Arquitetura do protótipo . . . 129
8.2 Disponibilidade . . . 131

9 Conclusão **133**

Referências Bibliográficas **135**

A Exemplos de redução do Sistema $\mathcal{W}$ **141**

Lista de Figuras

1.1 Um exemplo em Haskell. 9
1.2 Um exemplo em Haskell, usando tuplas. 10
1.3 Um exemplo em Haskell, usando construtores de dados. 10

2.1 Um fragmento de um programa em Haskell. 14
2.2 Duas implementações de números naturais (primeira parte) 19
2.3 Duas implementações de números naturais (segunda parte). 20

3.1 Um exemplo de registros em Haskell 24
3.2 Exemplo de registros em Haskell, após remoção de "açúcar sintático". 26
3.3 Exemplo de Haskell que mostra a ausência de polimorfismo de registro. 30

4.1 Um fragmento do semi-reticulado de tipos simples 41
4.2 Sistema de tipos do Sistema CT 48
4.3 Visão geral do processo de inferência de tipos do Sistema CT. 50
4.4 Regras de inferência para o cálculo da substituição principal. 51

5.1 Alguns símbolos Unicode. 57
5.2 A nova função *lcg*. 60
5.3 Sistema de tipos do Sistema CT com polimorfismo de registro. 66
5.4 Regras de satisfazibilidade com restrições de registro. 67

7.1 Sintaxe de usuário com sinônimos 122
7.2 Tradução de sintaxe de usuário para sintaxe interna. 123

Capítulo 1

Introdução

Uma das operações mais comuns em uma linguagem de programação é a construção de um objeto de dados composto a partir de um ou mais objetos de dados mais simples, bem como as operações que inspecionam, atualizam, removem e adicionam componentes a este objeto.

Tal construção é chamada por diferentes nomes em diferentes linguagens: **registros,** nas linguagens derivadas do Algol; **objetos**, em linguagens orientadas a objeto; **estruturas** em C, etc. Este documento segue a terminologia da linguagem Haskell, que chama estes objetos compostos de **registros**, onde cada componente do objeto é chamado um **campo** do registro e onde cada campo é identificado por um **nome de campo**.

A figura 1.1 mostra um exemplo de um programa em Haskell que constrói um registro e inspeciona e atualiza o valor de um campo.

A linguagem Haskell também provê outros dois mecanismos para a construção de valores compostos: tuplas e construtores de dados. Os programas apresentados nas figuras 1.2 e 1.3

```
module Main where
data InfoPessoal = InfoPess {
    nome :: String,
    idade :: Int,
    endereco :: String }
main =
  let  pi = InfoPess {
             nome = "Joao",
             idade = 27,
             endereco = "..." }
       pi' = pi { idade = 28 }
    in
       putStrLn $ "Minha idade é " ++ show (idade pi')
```

Figura 1.1: Um exemplo em Haskell.

```
module Main where
type InfoPess = (String,Int,String)
main =
  let pi = (“Joao”,27,“...”)
      pi’ = case pi of
              (n,_,a) → (n,28,a)
   in
      putStrLn $ “Minha idade é “ ++ show (case pi’ of (_,a,_) → a)
```

Figura 1.2: Um exemplo em Haskell, usando tuplas.

```
module Main where
data InfoPess = InfoPess String Int String
main =
  let pi = InfoPess “Joao” 27 “...”
      pi’ = case pi of
             (InfoPess n _ a) → InfoPess (n 28 a)
   in
      putStrLn $ “Minha idade é “ ++
         show (case pi’ of InfoPess _ a _ → a)
```

Figura 1.3: Um exemplo em Haskell, usando construtores de dados.

são versões reescritas do exemplo anterior, porém usando tuplas e construtores de dados, respectivamente.

A composição de objetos usando tuplas ou construtores de dados apresenta desvantagens em relação ao uso de registros, devido ao uso da posição de um componente para identificá-lo, ao invés de um rótulo:

- O programa é menos legível. O exemplo mostrado na figura 1.1 é bem menos legível que os outros.
- É mais susceptível a erros. O uso de identificação posicional é viável somente quando poucos componentes estão presentes no objeto composto. Como exemplo, considere a tarefa de lembrar a posição de um determinado componente em uma tupla com mais de 10 componentes.

Algumas desvantagens, que não são diretamente relacionadas ao uso de identificação posicional, mas são limitações da própria linguagem Haskell são:

- Não há nenhuma operação embutida na linguagem para inspecionar o valor de um componente, com exceção das funções `fst` e `snd`, que são funções de projeção para

duplas. Como visto nos exemplos anteriores, a inspeção de valores em tuplas com mais de dois elementos, ou em construtores de dados requer o uso de casamento de padrão (*pattern matching*), o que é bem mais complexo que uma simples chamada de função de projeção.

- Não há nenhuma operação embutida na linguagem para a atualização de um valor de um componente de uma tupla ou construtor de dados. Para atualizar um componente, é preciso fazer um casamento de padrão, associar uma variável a cada componente do objeto composto, e reconstruir o objeto composto a partir do novo valor do componente e das variáveis associadas a cada componente.

O sistema de registros da linguagem Haskell também tem sérias desvantagens, quando comparado a sistemas de registros usados em outras linguagens, ou a sistemas propostos pela comunidade de pesquisa. Estas desvantagens têm sido temas correntes nas listas de correio eletrônico usadas pela comunidade de programadores de Haskell, e são o tópico do capítulo 4, que comenta diversas propostas de correção destes problemas.

Este trabalho teve início com a seguinte questão: Como o sistema de restrições do Sistema CT pode ajudar o projeto de um sistema de registros? O resultado da pesquisa, apresentado neste documento, é um sistema de tipos que suporta operações polimórficas sobre registros, usando duas sintaxes para codificar os tipos de operações que usam polimorfismo de registro: a primeira usa o sistema de restrições do Sistema CT para codificar o polimorfismo de registro, descrita no capítulo 5, e a segunda usa uma sintaxe direcionada para o programador da linguagem, descrita no capítulo 7. O Sistema CT em si é apresentado no capítulo 4.

O capítulo 6 descreve a teoria de Conjuntos de Campos, que é usada para formular um algoritmo para construir a prova (ou contra-prova) de satisfazibilidade das restrições usadas para codificar o polimorfismo de registro e para calcular a generalização menos comum (*least common generalization*) das soluções de conjuntos formados por estas restrições.

Finalmente, uma breve descrição de um protótipo que implementa as idéias descritas neste documento é apresentada no capítulo 8.

Capítulo 2

Visão geral da linguagem Haskell

Este capítulo apresenta uma breve descrição da linguagem Haskell, com o propósito de familiarizar o leitor com a mesma. Leitores que já conhecem a linguagem podem saltar este capítulo, continuando a leitura no capítulo 3.

"Haskell é uma linguagem de propósito geral, puramente funcional, que incorpora muitas inovações recentes no seu projeto. Haskell provê funções de alta ordem, semântica não-estrita, sistema de tipos polimórfico com inferência e verificação estática, tipos de dados algébricos definidos pelo usuário, casamento de padrão, sintaxe especial para listas, um sistema de módulos, um sistema de E/S monádico e um rico conjunto de tipos de dados primitivos, incluindo listas, arranjos, inteiros de precisão fixa e arbitrária e números de ponto flutuante. Haskell é o ápice e a solidificação de vários anos de pesquisa em linguagens funcionais não-estritas." (O relatório Haskell 98 [18])

Comecemos com um pequeno fragmento de um programa em Haskell, mostrado na figura 2.1, que ilustra algumas características da linguagem.

Módulos

Programas em Haskell são organizados em **módulos**. Um módulo define valores, funções, tipos de dados, sinônimos de tipos, classes, etc. O módulo `SimpleSet`, apresentado na figura 2.1, é uma implementação de conjuntos usando listas, bem como de operações básicas sobre conjuntos: que são a constante `emptySet` e as funções `insert`, `remove`, `member`, `union` e `intersection`.

Anotações de tipos

No módulo `SimpleSet`, cada definição é precedida por uma *anotação de tipos*. Uma anotação de tipos é uma especificação parcial do significado de um símbolo.

A linguagem Haskell permite o uso de *polimorfismo paramétrico*. Todos os símbolos definidos no módulo `SimpleSet` possuem tipos polimórficos. Por exemplo, o tipo da constante `emptySet` – `[a]` – significa que este símbolo pode assumir diferentes tipos para cada

```
module SimpleSet where

emptySet :: [a]
emptySet = []

insert :: a → [a] → [a]
insert value set = value : set

remove :: Eq a ⇒ a → [a] → [a]
remove value set = [ v | v ← set, v /= value ]

member :: Eq a ⇒ a → [a] → [a]
member value set = not (null [ v | v ← set, v == value ])

union :: [a] → [a] → [a]
union [] set = set
union (value:setA) setB = insert value (union setA setB)

intersection :: Eq a ⇒ [a] → [a] → [a]
intersection [] set = []
intersection (value:setA) setB
   | member value setB = value : intersection setA setB
   | otherwise = intersection setA setB
```

Figura 2.1: Um fragmento de um programa em Haskell.

valor da *variável de tipo* `a`: `[Char]` (lista de caracteres), `[Integer]` (lista de inteiros), `[[Bool]]` (lista de listas de valores booleanos), etc.

Por outro lado, o tipo da constante `exSet`, definida abaixo, é um exemplo de um tipo *monomórfico*:

```
exSet :: [Integer]
exSet = [1,2,3,4,5]
```

Um tipo funcional especifica os tipos dos argumentos de uma função e de seu resultado. Um exemplo é a anotação de tipo do símbolo `insert`: `a→[a]→[a]`, que especifica que este símbolo é uma função cujos dois argumentos são um objeto de um tipo qualquer e uma lista de objetos deste tipo, e cujo resultado é também uma lista de objetos deste tipo. A função `insert` pode ser usada com o tipo `Int→[Int]→[Int]`, mas não com o tipo `Int→[Bool]→[Int]`.

Quando um símbolo é anotado com um tipo pelo programador, o compilador verifica se a definição dada para o símbolo pode efetivamente assumir o tipo anotado. Este processo

é chamado *verificação de tipos*. Como exemplo, considere a definição abaixo para a função `wrongFunction` anotada com o tipo `Integer`→`String`. Como o parâmetro `x` é usado como um valor booleano, o compilador identifica um erro durante a verificação de tipos:

```
wrongFunction :: Integer → String
wrongFunction x = if x then "true" else "false"

resultado do compilador:
    Couldn't match 'Bool' against 'Integer'
      Expected type: Bool
      Inferred type: Integer
    In the predicate expression: x
    In the definition of 'wrongFunction':
        wrongFunction x = if x then "true" else "false"
```

É importante notar que, com poucas exceções, anotações de tipos não são necessárias em programas em Haskell, pois o compilador é capaz de inferir um tipo para cada símbolo que representa todos os tipos que o programador poderia anotar para aquele símbolo. Este tipo é chamado o *tipo principal* do símbolo. O processo de cálculo dos tipos principais de um programa é chamado *inferência de tipos*.

Sintaxe de listas

A linguagem Haskell possui uma sintaxe especial para representar listas, que é usada nas definições feitas no módulo `SimpleSet` do exemplo anterior. O tipo de dados de uma lista, `[a]`, é um tipo de dados (*data type*) definido indutivamente como uma lista vazia, representada por `[]`, ou por uma lista, representada por `v : vl`, construída pela inserção de um valor `v`, de tipo `a`, na frente de uma lista `vl` de tipo `[a]`. Os símbolos `[]` e `(:)` são os *construtores de dados* de listas, cujos tipos são, respectivamente, `[a]` e `a` → `[a]` → `[a]`.

Valores de tipo lista podem ser escritos usando uma sintaxe especial, como mostra a definição da constante `exSet`: a notação `[1,2,3,4,5]` é simplesmente uma abreviação para `1 : (2 : (3 : (4 : (5 : []))))`.

As funções `remove` e `member` usam outra sintaxe especial para listas, que é análoga à sintaxe usada para representar conjuntos em teoria dos conjuntos. O significado das funções `remove` e `member` podem ser descritos pelas expressões correspondentes usando a sintaxe de teoria dos conjuntos:

$$\begin{aligned} \text{remove value set} &= \{v \mid bv \in \text{set} \wedge v \neq \text{value}\} \\ \text{member value set} &= \{v \mid v \in \text{set} \wedge v = \text{value}\} \neq \emptyset \end{aligned}$$

Finalmente, a linguagem Haskell também inclui uma sintaxe especial para a definição de sequências aritméticas:

- `[10 .. 20]`: lista de números naturais de 10 a 20.
- `[2 ..]`: lista de números naturais acima de 1.
- `[1,3 ..]`: lista de números naturais ímpares.

Casamento de padrão

As funções `union e intersection`, definidas no módulo `SimpleSet`, são exemplos de definições feitas usando *casamento de padrão*, que define um valor através de diversas equações alternativas, onde cada alternativa representa um padrão de valor para o argumento da função. As definições de ambas as funções possuem duas alternativas: a primeira para listas vazias, que casam com o padrão `[]`, e a segunda para listas não-vazias, que casam com o padrão `(v : vl)`.

O casamento de padrão é realizado na ordem textual em que as alternativas aparecem no programa.

Recursão

Em linguagens funcionais, laços e repetições são construídos a partir de *definições recursivas* de funções. As funções `union` e `intersection`, do módulo `SimpleSet`, são exemplos de funções recursivas. Definições que usam recursão e casamento de padrão usualmente são casos de *indução estrutural*.

Guardas

A definição da função `intesection` é um exemplo de uma definição que faz uso de *expressões com guardas*, que permite a especificação de definições alternativas, cada qual relacionada com uma expressão booleana (guarda). Definições usando guardas são similares a definições por caso da matemática, o que pode ser ilustrado comparando-se a definição da função `intersection` com a seguinte definição:

$$\begin{aligned} \emptyset \cap \mathrm{X} &= \emptyset \\ (\{v\} \cup X) \cap Y &= \begin{cases} \{v\} \cup (X \cap Y) & \text{se } v \in Y \\ X \cap Y & \text{se } v \notin Y \end{cases} \end{aligned}$$

O módulo `NaturalNumbers`, definido nas figuras 2.2 e 2.3, apresenta duas implementações distintas de números naturais e exemplifica outras duas características importantes da linguagem Haskell: declarações de tipos de dados (*datatypes*) e de classes de tipos (*type classes*).

Tipos de dados

As definições de tipos de dados presentes no módulo `NaturalNumbers` revelam a sintaxe usada para definir um novo tipo de dados em um programa. A linha

```
data UnaryNatural = UnaryZero | UnarySucc UnaryNatural
```

define um tipo de dados que codifica números naturais usando uma representação unária.

Uma definição de um tipo de dados consiste em um conjunto de definições de construtores de dados, separados pelo símbolo '|'. Cada construtor de dados é identificado por rótulo único no escopo do programa, que o distingue de outros construtores de dados e possui como especificação uma sequência de tipos que representam os tipos dos parâmetros deste construtor.

Um número natural representado por `UnaryNatural` é ou o número zero (`UnaryZero`) ou o sucessor de outro número natural unário (`UnarySucc`).

As outras duas definições de tipos do módulo `NaturalNumbers` também codificam números naturais, porém usando uma representação binária, com o dígito mais significante para a direita:

```
data BinaryNatural = BinNone | BinMore BinDigit BinaryNatural

data BinDigit = Zero | One
```

Um tipo de dados pode ser parametrizado. Um bom exemplo de um tipo de dados parametrizado é o tipo lista:

```
data Lista a = ListaVazia | ListaNaoVazia a (Lista a)
```

Outro exemplo é o tipo de uma árvore binária:

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

Um tipo de dados pode ter mais de um parâmetro, como o seguinte tipo de dicionário:

```
data DictTree key value = DictLeaf |
          DictNode key value (DictTree key value) (DictTree key value)
```

Um tipo de dados parametrizado é também chamado um tipo de dados *polimórfico.*

Classes de tipos

O sistema de *classes de tipos* é uma característica importante da linguagem Haskell. Através de classes de tipos, um programador pode definir símbolos *sobrecarregados*, que pode ter diferentes definições para tipos de dados diferentes.

Restrições de classes de tipos podem aparecer em um tipo polimórfico, representando uma restrição sobre o conjunto de possíveis valores de tipos que o tipo polimórfico pode assumir. Por exemplo, o tipo da função `calcPoly`, definida no módulo `NaturalNumbers`, `Natural a` ⇒ `Polynomial a` → `a`→ `a`, especifica que `calcPoly` pode ser usado para calcular o valor de um polinômio sobre qualquer tipo para o qual a restrição `Natural a` vale.

Para entender a restrição `Natural a`, precisamos considerar a definição da classe de tipos 'Natural a':

```
class Natural a where
  zero :: a
  suc :: a → a
  add :: a → a → a
  mul :: a → a → a
```

Esta classe especifica que para satisfazer a restrição `Natural a` é necessário que hajam implementações para os símbolos `zero`, `suc`, `add`, e `mul` para o tipo `a`. Estas implementações formam a *instância* da classe de tipos e são definidas através de *declarações de instâncias.*

O módulo `NaturalNumbers` inclui duas declarações de instâncias da classe `Natural`: uma para a implementação unária (`UnaryNatural`) e outra para a implementação binária (`BinaryNatural`). Portanto, a função `calcPoly` pode ser usada com os tipos

```
Polynomial UnaryNatural → UnaryNatural → UnaryNatural e

Polynomial BinaryNatural → BinaryNatural → BinaryNatural
```

Alguns outros exemplos de classes de tipos frequentemente usados em programas Haskell são:

- `Eq a`: O operação de igualdade (`==`) está definida para o tipo `a`.
- `Ord a`: A restrição `Eq a` tem de ser satisfeita, e a operação de menor-ou-igual (`<=`) deve estar definida para o tipo `a`.
- `Num a`: Os seguintes símbolos devem estar definidos:
  - `(+): :  a → a → a`
  - `(*): :  a → a → a`
  - `(-): :  a → a → a`
  - `negate: :  a → a`

```
module NaturalNumbers here

class Natural a where
  zero :: a
  suc :: a → a
  add :: a → a → a
  mul :: a → a → a

data UnaryNatural = UnaryZero | UnarySucc UnaryNatural

instance Natural UnaryNatural where
  zero = UnaryZero

  suc x = UnarySucc x

  add UnaryZero v = v
  add (UnarySucc v1) v2 = UnarySucc (add v1 v2)

  mul UnaryZero v = UnaryZero
  mul (UnarySucc v1) v2 = add v2 (mul v1 v2)
```

Figura 2.2: Duas implementações de números naturais (primeira parte)

- `abs: :  a → a`
- `signum: :  a → a`
- `fromInteger: :  Integer → a`

Classes de tipos com múltiplos parâmetros não fazem parte do padrão Haskell 98 da linguagem, porém diversos compiladores e interpretadores permitem o seu uso como uma extensão da linguagem, como o compilador GHC e o interpretador Hugs.

Um bom exemplo de código que usa classes com múltiplos parâmetros é a *Biblioteca de Transformadores Monádicos*, inclusa na biblioteca do compilador GHC.

Dependências funcionais

Se classes de tipos são usadas para restringir os possíveis valores que certas variáveis de tipos podem assumir em tipos polimórficos, *dependências funcionais* são usadas para restringir as possíveis instâncias que um programador pode declarar para uma certa classe de tipos. O código seguinte é um fragmento da Biblioteca de Transformadores Monádicos, que define classes de tipos com múltiplos parâmetros que fazem uso de dependências funcionais.:

```
class Monad m ⇒ MonadState s (m::* → *) | m → s where
```

```
data BinaryNatural = BinNone | BinMore BinDigit BinaryNatural

data BinDigit = Zero | One

instance Natural BinaryNatural where
  zero = BinNone

  suc BinNone = BinMore One BinNone
  suc (BinMore Zero r) = BinMore One r
  suc (BinMore One r) = BinMore Zero (suc r)

  add BinNone r  = r
  add r  BinNone = r
  add (BinMore Zero r1) (BinMore d r2) = BinMore d (add r1 r2)
  add (BinMore d r1) (BinMore Zero r2) = BinMore d (add r1 r2)
  add (BinMore _ r1) (BinMore _ r2) = BinMore Zero (suc (add r1 r2))

  mul BinNone r = BinNone
  mul r  BinNone = BinNone
  mul (BinMore Zero r1) r2 = BinMore Zero (mul r1 r2)
  mul (BinMore One r1) r2  = add r2  (BinMore Zero (mul r1 r2))

type Polynomial a = [a]

calcPoly :: Natural a ⇒ Polynomial a → a → a
calcPoly [] _ = zero
calcPoly (coef:rest) x = add (mul coef x) (mul x (calcPoly rest x))
```

Figura 2.3: Duas implementações de números naturais (segunda parte).

```
  get :: m s
  put :: s → m ()
```

O termo `(m::*→*)` é uma anotação de *kind*. Ele especifica que a variável `m` não representa um tipo, mas sim um construtor de tipos que têm um parâmetro de *kind* `*` (o *kind* de tipos).

A dependência funcional (`m → s`) restringe as possíveis instâncias da classe `MonadState` ao requerer que, para cada construtor de tipos `m`, deva existir no máximo uma instância. Devido a esta restrição, as seguintes instâncias não poderiam ser declaradas pelo programador:

```
MonadState Int MyStateMonad
MonadState String MyStateMonad
```

Apesar de que estas outras poderiam:

```
MonadState Int []
MonadState String MyStateMonad
```

Isto ocorre porque, no primeiro caso, haveriam dois tipos distintos para a variável `s` (`Int`, `String`) para o mesmo valor da variável `m`, contrariando assim a dependência funcional declarada na definição da classe (`m`→`s`)..

Dependências funcionais também são uma extensão ao padrão Haskell 98, mas como pode ser facilmente verificado pela leitura da lista de discussão de Haskell [1], ou pelos comentários feitos por usuários do compilador GHC na pesquisa de 2005 [3], dependências funcionais e classes de tipos com múltiplos parâmetros parecem ser indispensáveis para muitos projetos.

A sintaxe de Haskell para registros é apresentada no próximo capítulo, que mostra as limitações do sistema de registros de Haskell.

Capítulo 3

Registros em Haskell

A linguagem Haskell possui uma sintaxe especial para operações que envolvem registros, como ilustrado pelo exemplo da figura 3.1.

3.1 Declarações de registros

A primeira declaração do módulo `RecordExample` mostrado na figura 3.1 é uma declaração de registro:

```
data FileInformation = FI {
        fileName :: String,
        accessFlags :: [Bool],
        accessTime :: Integer
    }
```

Um registro é sempre associado a um construtor de dados em Haskell. Dentro dos colchetes estão as declarações dos campos do registro, que especificam o nome e o tipo de cada campo do registro. O nome de um campo não pode ser o nome de nenhum símbolo visível no escopo do programa, nem pode ser o nome de outro campo presente em um registro definido em outro tipo de dados. O seguinte caso provocaria uma mensagem de erro do compilador:

```
fileName = ‘‘afilename’’

data FileInformation = FI {
        fileName :: String, ...
```

No entanto, registros definidos no mesmo tipo de dados, porém em construtores de dados diferentes, podem ter campos com o mesmo nome, desde que os tipos associados sejam os mesmos:

```
module RecordExample where

data FileInformation = FI {
        fileName :: String,
        accessFlags :: [Bool],
        accessTime :: Integer
    }

someFileInfo :: FileInformation
someFileInfo = FI { fileName = "README.TXT",
                    accessFlags = [True | _ ← [1 .. 8]],
                    accessTime = 1234559 }

type FlagNum = Int

setFlag :: FlagNum → Bool → [Bool] → [Bool]
setFlag 0 v (_:r) = v : r
setFlag n v (s:ss) = s : setFlag (n-1) v ss

enableFlag :: FlagNum → FileInformation → FileInformation
enableFlag n fi =
          fi { accessFlags = setFlag n True (accessFlags fi) }

disableFlag :: FlagNum → FileInformation → FileInformation
disableFlag n fi@(FI { accessFlags = fl }) =
          fi { accessFlags = setFlag n False fl }
```

Figura 3.1: Um exemplo de registros em Haskell

```
data FileInformation = FI {
        fileName :: String,
        accessFlags :: [Bool],
        accessTime :: Integer
    } | SpecialFile { fileName :: String, otherInfo :: OtherInfo }
```

Construção de registros

Um valor de tipo registro pode ser definido como mostrado na definição do símbolo `someFileInfo`:

```
someFileInfo = FI { fileName = "README.TXT",
                    accessFlags = [True | _ ← [1 .. 8]],
```

```
                    accessTime = 1234559 }
```

O construtor de dados deve ser seguido por uma lista correspondente de atribuições a campos do registro correspondente. Cada campo declarado do registro deve ter uma atribuição correspondente.

Funções de projeção

Quando um registro é declarado em Haskell, uma função de projeção é automaticamente declarada para cada campo do registro. Cada função de projeção tem o mesmo nome do campo correspondente e tem o tipo:

`nomedocampo ::  TipoDadosRegistro` $p_1\ p_2 \cdots p_n \rightarrow$ `tipo do campo`

onde $p_1, \ldots, p_i$ são os parâmetros do tipo de dados.
A definição da função `enableFlag` demonstra o uso de uma função de projeção:

```
enableFlag n fi =
          fi { accessFlags = setFlag n True (accessFlags fi) }
```

Casamento de padrão de registros

É possível fazer casamento de padrão com um ou mais campos de um registro. O padrão deve começar com o nome do construtor de dados correspondente e deve ser seguido dos padrões que deverão ser casados com cada campo, cada um precedido pelo nome do campo correspondente, como mostra o seguinte exemplo:

```
case varReg of
   TipoDadosReg { campo1 = padCampo1,
                  campo2 = padCampo2, ...} → ...
```

A função `disableFlag` mostra o uso de casamento de padrão para registros:

```
disableFlag n fi@(FI { accessFlags = fl }) = ...
```

Atualização de registros

A última sintaxe de Haskell relacionada a registros é a operação de atualização de registros. Dada uma expressão que resulta em um valor de um tipo de dados com um registro, é possível construir um novo valor com um ou mais campos do registro alterados. Isto é feito acrescentando-se à expressão uma lista de atribuição de campos, de forma semelhante à sintaxe de construção de registros, porém sem a exigência de se especificar todos os registros. As definições das funções `enableFlags` e `disableFlag` fazem uso da sintaxe de atualização de registros:

```
module RecordExample where

data FileInformation = FI String [Bool] Integer

fileName (FI x _ _) = x
accessFlags (FI _ x _) = x
accessTime (FI _ _ x) = x

someFileInfo :: FileInformation
someFileInfo = FI "README.TXT" [True | _ ← [1 .. 8]] 1234559

type FlagNum = Int

setFlag :: FlagNum → Bool → [Bool] → [Bool]
setFlag 0 v (_:r) = v : r
setFlag n v (s:ss) = s : setFlag (n-1) v ss

enableFlag :: FlagNum → FileInformation → FileInformation
enableFlag n fi =
  FI (fileName fi) (setFlag n True (accessFlags fi)) (accessTime fi)

disableFlag :: FlagNum → FileInformation → FileInformation
disableFlag n fi@(FI _ fl _) =
  FI (fileName fi) (setFlag n False fl) (accessTime fi)
```

Figura 3.2: Exemplo de registros em Haskell, após remoção de "açúcar sintático".

```
enableFlag n fi =
          fi { accessFlags = setFlag n True (accessFlags fi) }

disableFlag ... = fi { accessFlags = setFlag n False fl }
```

Em ambos os casos, a sintaxe de atualização de registros é usada para mudar o valor do campo `accessFlags` do valor denotado pela variável `fi`.

Registros em Haskell são "açúcar sintático"

Em Haskell, um registro não é um tipo primitivo, mas apenas uma abreviação para uma declaração de um construtor de dados e de funções de projeção correspondentes. Para entender como isto é feito, veja o exemplo apresentado na figura 3.2, que mostra como o código da figura 3.1 é visto pelo compilador Haskell.

A transformação procede da seguinte maneira:

1. Cada construtor de dados de registro é transformado em uma declaração normal de construtor de dados, com o número e tipos de parâmetros equivalentes ao número e tipos dos campos do registro. Portanto, a declaração de tipo de dados

```
data FileInformation = FI {
        fileName :: String,
        accessFlags :: [Bool],
        accessTime :: Integer
    }
```

   é reescrita como:

```
data FileInformation = FI String [Bool] Integer
```

1. Para cada campo, uma função de projeção é definida:

```
fileName (FI x _ _) = x
accessFlags (FI _ x _) = x
accessTime (FI _ _ x) = x
```

2. Um casamento de padrão de registro é transformado em um casamento de padrão normal do construtor de dados relacionado. Os campos que não foram relacionados no casamento de padrão são substituídos por '_' nos padrões de seus parâmetros correspondentes:

```
disableFlag n fi@(FI { accessFlags = fl }) = ...
```

   é reescrito como:

```
disableFlag n fi@(FI _ fl _) = ...
```

1. Uma atualização de registro é simplesmente um casamento de padrão seguido da reconstrução do valor:

```
 fi { accessFlags = setFlag n True (accessFlags fi) }
```

   é reescrito como:

```
case fi of
  FI fiFname _ fiAccessTime →
    FI fiFname (setFlag n True (accessFlags fi)) fiAccessTime
```

3.2 Problemas com o projeto atual de registros em Haskell

Apesar do sistema de registros de Haskell ser útil, programadores que o usam vão invariavelmente esbarrar em algumas de suas limitações. A seguinte lista é baseada em críticas apresentadas na lista de discussão de Haskell, com alguns acréscimos do próprio autor:

Sintaxe e semântica pouco intuitivas

Tipos de dados algébricos possuem dois construtores de dados primitivos: a *soma* e o *produto*. O primeiro é usado para representar informação *alternativa,* enquanto que o último é usado para representar informação simultânea. Estas noções são ortogonais entre si e são "duais" em formulações de tipos algébricos usando a Teoria de Categorias [34].

As definições destas duas construções na Teoria de Categorias especificam um número de funções, uma para cada alternativa de uma *soma* e uma para cada componente de um *produto*. No caso de somas, estas funções são chamadas *injeções*, e transformam um objeto do tipo de uma alternativa no tipo da soma, enquanto que no caso dos produtos estas funções são chamadas *projeções* e transformam um objeto do tipo do produto em um de seus componentes.

Normalmente, estas funções são identificadas por um número, de 1 ao número de alternativas ou componentes da soma ou produto, respectivamente. Uma definição de tipo de dados em Haskell com mais de um construtor de dados é na verdade uma definição de um tipo soma, porém com as suas alternativas identificadas por um nome de um construtor de dados ao invés de um número.

O conceito correspondente, no caso de um produto, é o registro, que identifica seus componentes por nomes ao invés de números. Portanto, parece ser pouco intuitivo que para definir-se um tipo relacionado a produtos (registros) seja necessário usar a sintaxe relacionada a somas (*datatypes*).

Idealmente, um registro deveria ter um tipo específico (como tuplas), e não ter de ser associados a uma definição de tipo de dados.

Registros em Haskell são "pesados"

Devido à codificação de valores de registro como valores de um tipo de dados particular declarado pelo programador, registros em Haskell precisam ser declarados antes de serem usados, o que pode desestimular o seu uso. Por exemplo, suponha que o programador deseje substituir a passagem de um parâmetro à uma função através de uma tupla por um registro:

```
func (parm1, parm2, parm3, parm4, parm5) ... =
                    ↓
func (Reg { parm1 = parm1, parm2 = parm2, parm3 = ...}) ... =
```

Esta substituição a princípio parece ser interessante, de um ponto vista de boas práticas de programação, pois substitui o acesso a componentes da tupla por posição pelo acesso por nome, o que facilita o entendimento do programa e reduz a possibilidade de erro do programador.

No entanto, para efetivá-la, o programador precisa declarar pelo menos um tipo de dados e um construtor de dados, o que "poluiria" o espaço de nomes de tipos e de construtores de dados:

```
data TipoReg = Reg { parm1 :: ..., parm2 :: ... }
```

Um projeto alternativo, que não possui a limitação citada é o da linguagem Standard ML [28]:

```
fun decTTL { payload, ttl, sourceAddress, destAddress } =
           { payload = payload, ttl = ttl - 1,
             sourceAddress = sourceAddress,
             destAddress = destAddress }
```

Como Standard ML não exige que os tipos registro sejam declarados, e associados a um nome, antes de seu uso, podem ser usados sem poluir o espaço de nomes.

Nomes de campos não podem ser usados em tipos registro diferentes

Outra limitação do sistema de registros de Haskell é que registros definidos em tipos de dados distintos não podem usar os mesmos nomes de campos, ou seja, o programador também tem que se preocupar com a poluição do espaço de nomes de campos de registros e de funções.

Para prevenir o conflito entre nomes de campos de diferentes registros, o programador tem que seguir alguma política, como a de declarar cada registro em um módulo diferente e sempre fazer referência ao mesmo usando um nome *qualificado*, prefixando o nome do campo com o nome ou apelido do módulo onde o registro foi declarado.

A maioria das linguagens permite que registros compartilhem nomes de campos, incluindo linguagens tradicionais, como C, Pascal e também outras linguagens funcionais, como Standard ML. Linguagens orientadas a objeto também não apresentam esta limitação, pois duas classes podem ter atributos e métodos com o mesmo nome.

Não há suporte para polimorfismo de registro

Como Haskell não permite o compartilhamento de campos entre tipos de registro, não é possível expressar polimorfismo de registro, pois cada operação de registro sempre se refere a um tipo registro específico.

Como exemplo, considere a seguinte função, escrita na linguagem SML# [31], que é uma variação de Standard ML que permite polimorfismo de registro:

```
class HasFieldTTL a where
  projTTL :: a → Int
  updTTL  ::  Int → a → a
data IpPacket = IP { ipTTL :: Int,
                     ipAddress :: (Int,Int,Int,Int), ... }
data AtmPacket = ATM { atmTTL :: Int, atmAddress :: (Int,Int), ... }
instance HasFieldTTL IpPacket where
  projTTL = ipTTL
  updTTL v x = x { ipTTL = v }
instance HasFieldTTL AtmPacket where
  projTTL = atmTTL
  updTTL v x = x { atmTTL = v }
fun decTTL packet = updTTL (projTTL packet - 1) packet
```

Figura 3.3: Exemplo de Haskell que mostra a ausência de polimorfismo de registro.

```
fun decTTL (packet as {ttl, ...}) = packet # { ttl = ttl - 1 }
v1 = decTTL { ttl = 10, ipAddress = (10,1,1,1), ... }
v2 = decTTL { ttl = 13, atmAddress = (3,5), ... }
```

O primeiro parâmetro desta função pode ser qualquer registro que tenha um campo com nome `ttl` e com tipo inteiro. Para expressar a mesma função em Haskell, o programador teria que declarar uma classe de tipos para representar registros que possuem o campo `ttl`, com métodos que façam a atualização e a projeção do campo, como mostra a figura 3.3.

Além de atualização e projeção, outras operações com polimorfismo de registro também são interessantes:

- *extensão de registro*: estende um registro com novo campo.
- *restrição de registro*: remove um campo de um registro.
- *concatenação de registro*: combina dois registros com nomes de campos distintos em um só.

Nenhuma destas operações é possível em Haskell.

Não há uma função de atualização de registros

Enquanto que para cada campo de um registro a linguagem Haskell provê uma função de projeção, facilitando a inspeção de um campo de um registro, o mesmo não ocorre para a atualização.

Seria interessante que cada operação de registro tivesse uma função correspondente. Para que isso ocorra, é conveniente separar o espaço de nomes de funções declaradas pelo usuário do espaço de nomes de funções relativas às operações de registro.

3.3 Alternativas

Esta seção apresenta uma revisão de diversas propostas que têm sido feitas para corrigir as limitações do sistema de registros de Haskell. Com uma única exceção, todas as propostas exigem alterações na sintaxe da linguagem, em seu sistema de tipos e seu sistema de classes.

Variáveis de linha e predicados *omite*

O sistema de tipos proposto em [11] usa variáveis de linha (*row variables*) [7] bem como um predicado especial *omite* (*lack*) sobre linhas para definir um sistema de registros com algumas operações polimórficas.

A extensão Trex ("typed records with extension") [10] implementada no interpretador Hugs é baseada no sistema proposto em [11] e representa a primeira definição de um sistema de registro para Haskell que permite operações com polimorfismo de registro. As operações básicas de registro fornecidas pelo sistema, com os seus respectivos tipos, são:

- (*projeção*) $(_.\mathbf{l}) \;::\; (r\backslash\mathbf{l}) \Rightarrow \mathrm{Rec}\{\!|\, \mathbf{l} : \alpha | r \,|\!\} \rightarrow \alpha$
- (*restrição*) $(_ - \mathbf{l}) \;::\; (r\backslash\mathbf{l}) \Rightarrow \mathrm{Rec}\{\!|\, \mathbf{l} : \alpha | r \,|\!\} \rightarrow \mathrm{Rec}\, r$
- *(extensão)* $(\mathbf{l} = _|_) \;::\; (r\backslash\mathbf{l}) \Rightarrow \alpha \rightarrow \mathrm{Rec}\, r \rightarrow \mathrm{Rec}\{\!|\, \mathbf{l} : \alpha | r \,|\!\}$

Jones et al. [16] propôs um sistema similar, que modifica a sintaxe de Haskell e seu sistema de tipos para permitir o uso de registros leves (*lightweight records*). Juntamente com a proposta são identificados também alguns possíveis problemas, em especial a combinação do sistema de classes tipos e variáveis de linha.

Rótulos de primeira classe

Leijen [24] propôs um sistema de tipos onde rótulos (nomes) de campos são objetos de primeira classe na linguagem. O resultado é um sistema de tipos expressivo, que permite a passagem de rótulos como parâmetros em chamadas de funções.

Cada rótulo é codificado como um tipo de dados distinto, com um único construtor de dados sem parâmetros. Os tipos de dados de rótulos possuem um *kind* diferente dos outros tipos, que é chamada o *kind* de rótulos. Os tipos das operações de registro básicas são:

- (*projeção*) $(_._) \;::\; (r\backslash l) \Rightarrow \{\; l \;::\; \alpha | r \;\} \rightarrow \mathrm{Lab}\, l \rightarrow \alpha$
- (*restrição*) $(_ - _) \;::\; (r\backslash l) \Rightarrow \{\; l \;::\; \alpha | r \;\} \rightarrow \mathrm{Lab}\, l \rightarrow \{r\}$
- (extensão) $(_ = _|_) \;::\; (r\backslash l \Rightarrow \mathrm{Lab}\, l \rightarrow \alpha \rightarrow \{r\} \rightarrow \{\; l \;::\; \alpha | r \;\}$

O sistema de Leijen é semelhante, em alguns aspectos, ao sistema proposto neste documento:

- Um predicado *lacks* é usado para expressar a ausência de campos de um registro.

- Um predicado especial é usado para expressar a relação entre dois tipos de registro. No caso do sistema de Leijen, ele é chamado de predicado de igualdade de linhas, enquanto que este documento usa o predicado *extends*.
- A proposta de Leijen especifica regras para as fases de aprimoramento (*improvement*) e de simplificação (*simplification*) do algoritmo definido em [17]. O algoritmo definido neste documento cumpre um papel semelhante ao algoritmo de aprimoramento.

A proposta do autor difere em que variáveis de linha e variáveis de rótulo não são usadas e rótulos não são objetos de primeira classe na linguagem.

Rótulos com escopo (*scoped labels*)

Um sistema semelhante ao anterior é proposto por Leijen em [25], diferindo do anterior ao permitir que dois campos distintos do mesmo registro tenham o mesmo nome. Leijen mostrou como este sistema pode ser usado para codificar módulos de primeira classe, quando usado em um sistema de tipos impredicativo.

Este sistema é implementado pelo compilador Morrow [26].

Tipos "fantasma" (*phantom types*)

A última proposta comentada é o sistema HList [23], que usa *tipos fantasma*, classes com múltiplos parâmetros e dependências funcionais para codificar estruturas de dados heterogêneas, incluindo registros. Este trabalho é a base da biblioteca OO'Haskell [22].

Esta codificação requer apenas algumas extensões presentes na maioria dos compiladores e interpretadores Haskell. Porém, a sintaxe das operações de registro é complexa e os tipos resultantes são difíceis de serem entendidos. Além disso, as operações de registro são bastante ineficientes, pois o acesso a campos de um registro é feito em tempo proporcional ao número de campos do mesmo.

Capítulo 4

Sistema CT

4.1 Classes de tipos de Haskell

Polimorfismo é um alvo frequente de projetos de linguagens de programação estaticamente tipadas. Uma construção polimórfica, seja ela um valor, função, método, procedimento, etc., é uma construção à qual podemos atribuir diversos tipos.

Os exemplos mais antigos de construções polimórficas de linguagens de programação são as operações aritméticas do FORTRAN, que podem assumir diversos tipos numéricos. Linguagens estaticamente tipadas modernas em geral permitem alguma forma de polimorfismo. Linguagens orientadas a objeto permitem o *polimorfismo limitado* (*bounded polymorphism* [4]). A família de linguagens Damas-Milner [8], à qual pertencem boa parte das linguagens funcionais estaticamente tipadas, permite o *polimorfismo paramétrico*. A maioria das linguagens permitem o polimorfismo *ad-hoc*.

Estas formas de polimorfismo são ortogonais entre si e muitas vezes são usadas ao mesmo tempo, como, por exemplo, em linguagens orientadas a objeto, que geralmente permitem o uso de subtipagem, através de herança (polimorfismo limitado) e sobrecarga de métodos (polimorfismo *ad-hoc*).

O polimorfismo *ad-hoc* permite que um símbolo possa assumir diversos tipos. O conjunto de tipos que estes símbolos podem assumir é muitas vezes determinado no projeto da linguagem, como os literais numéricos e operações aritméticas em C ou Java, e em outras linguagens pode ser estendido pelo programador, como em C++. Tais símbolos são chamados símbolos *sobrecarregados*. Um tratamento mais amplo sobre sobrecarga de operações pode ser encontrado em [6].

Boa parte das linguagens sobrecarrega alguns de seus símbolos, usualmente as operações aritméticas e os operadores de comparação.

No entanto, ao contrário das outras formas de polimorfismo, o polimorfismo *ad-hoc*, como sugere o próprio nome, têm sido um aspecto arbitrário e sem embasamento teórico na maioria das linguagens.

Não é surpresa, portanto, que teóricos de linguagem de programação têm mostrado um interesse em particular neste assunto. O Sistema de Classes da linguagem Haskell é um

resultado desta pesquisa.

Haskell permite que o programador defina símbolos sobrecarregados *dependentes de contexto*, que são *métodos* especificados em uma *classe de tipos*. A classe de tipos é parametrizada por uma variável de tipo, o *parâmetro da classe*, e serve como especificação do tipo de símbolos sobrecarregados, que podem então ser declarados pelo usuário através de *instâncias de classe*. Cada instância corresponde à sobrecarga do conjunto de símbolos especificados na classe de tipos, para um valor específico do parâmetro da classe.

Faz-se necessária a declaração de uma classe de tipos em Haskell antes de se fazer a sobrecarga dos símbolos correspondentes porque o tipo de sobrecarga permitida em Haskell é a sobrecarga de *mundo aberto*.

O seguinte exemplo mostra a declaração da classe `Eq` e de uma instância da classe (para o valor do parâmetro `NatNum`):

```
class Eq a where
  (==) :: a → a → Bool
  (/=) :: a → a → Bool
equal3 :: Eq a ⇒ a → a → a → Bool
equal3 x y z = (x == y) && (y == z)
data NatNum = NatZero | NatSucc NatNum
instance Eq NatNum where
  NatZero == NatZero = True
  NatSucc n1 == NatSucc n2 = n1 == n2
  _ == _ = False

  n1 /= n2 = not (n1 == n2)
```

A classe `Eq` especifica que o programador deve assumir que os símbolos (`==`) e (`/=`) estão sobrecarregados e que os tipos das sobrecargas precisam ser instâncias de `a→a→Bool`. Com estas restrições, o compilador pode inferir o tipo que foi anotado para a função `equal3`. Esse tipo é polimórfico, porém apresenta uma *restrição*: ele só pode assumir o tipo `a→a→a→Bool` para valores de `a` tais que existam os símbolos correspondentes a `Eq a` no contexto. Em particular, ele pode assumir o valor `NatNum → NatNum → NatNum → Bool`.

Classes podem ser relacionadas entre si como superclasses e subclasses:

```
class Eq a ⇒ Num a where
  (+) :: a → a → a
  (-) :: a → a → a
  (*) :: a → a → a
```

Neste caso, temos que para que um tipo `a` seja considerado um elemento da class `Num`, é preciso que haja uma sobrecarga correspondente para o símbolos (`+, -, *`), mas também que haja as sobrecargas correspondentes a `Eq a`.

O sistema de sobrecarga proposto pelo Sistema CT apresenta uma visão alternativa, baseada na suposição de que o programador não deveria ter que declarar uma classe de

tipos (inventando um nome para a classe, decidindo quais são os métodos e tipos dos mesmos) somente para sobrecarregar símbolos.

Ao invés disso, o Sistema CT permite que o programador defina operações sobrecarregadas a medida que for necessário, usando *tipos restringidos* (*constrained types*) no lugar de classes de tipos.

Um tipo restringido é um tipo parametrizado juntamente com um conjunto de restrições sobre as variáveis deste. Este conjunto restringe os tipos para os quais estas variáveis podem ser instanciadas, na forma de exigências sobre a existência de certos símbolos no contexto onde o tipo será instanciado.

Usando tipos restringidos, a função `equal3` teria o seguinte tipo:

```
equal3 :: {(==) a → a → Bool} . a → a → a → Bool
```

Este tipo significa que a função `equal3` pode ser usada com parâmetros de tipo `a` desde que em um contexto onde a operação de igualdade (`==`) esteja definida para o tipo `a`.

Este capítulo apresenta uma breve descrição do sistema de tipos do Sistema CT, no qual o sistema de registro proposto neste documento foi baseado.

4.2 Sintaxe

Termos

O Sistema CT usa a seguinte linguagem-núcleo de termos:

Variáveis de termos (ou símbolos)	$x \in \mathbf{X}$	
Programas	$p \in \mathbf{P}$	$::= e \mid \texttt{leto}\ x = e\ \texttt{in}\ p$
Expressões	$e \in \mathbf{E}$	$::= x \mid \lambda x.\, e \mid e\, e' \mid \texttt{let}\ x = e\ \texttt{in}\ e'$

A sintaxe livre de contexto descrita acima é, basicamente, a linguagem *core-ML* [27, 8, 29] estendida com a construção `leto`, que introduz definições sobrecarregadas no escopo mais externo do programa (todos os símbolos sobrecarregados são definidos no escopo mais externo do programa).

Tipos e *kinds*

Tipos e *kinds* são expressos usando a seguinte linguagem:

Kinds	$K \in \mathbf{Kind}$	$::= \star \mid K \to K$	
Variáveis de tipos	$\alpha, \beta, \gamma \in \mathbf{V}$		
Conjuntos de variáveis de tipos	$V, \bar{\alpha}, \bar{\beta}, \bar{\gamma} \subseteq \mathbf{V}$		
Construtores de tipos	$C \in \mathbf{C}$		
Expressões de tipo simples	$\tau \in \mathbf{T}$	$::= \alpha \mid C \mid \tau_1\, \tau_2$	
Conjuntos de restrições	$\kappa \in \mathbf{K}$	$::= \{c_1, \ldots, c_n\}$	$n \geq 0$
Restrições	c	$::= x : \tau$	
Tipos	$\sigma \in \mathbf{J}$	$::= \forall \bar{\alpha}.\kappa.\tau$	

Tipos simples e *kinds*

Um *kind* é uma propriedade de expressões de tipo simples. O *kind* de variáveis de tipo de construtores de tipo é dado, respectivamente, pelas funções $\mathrm{kind}_{\mathbf{V}} : \mathbf{V} \to \mathbf{Kind}$ e $\mathrm{kind}_{\mathbf{C}} : \mathbf{C} \to \mathbf{Kind}$. Estas funções induzem famílias de variáveis de tipo e de construtores, indexadas por *kind*:

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

Um índice superior k é usado para identificar o *kind* de variáveis de tipo e construtores:

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

O *kind* de expressões de tipo simples é dado pela função parcial $\mathrm{kind}_{\mathbf{T}} : \mathbf{T} \to \mathbf{Kind}$, definida como:

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

Consideraremos somente expressões de tipo que forem *bem formadas*, isto é, aquelas que têm um *kind* definido. Por isso, o conjunto $\mathbf{T}$ só irá incluir expressões bem formadas. Por definição, o *kind* de tipos simples é $\star$.

Como o usual, a sintaxe infixa é usada para expressões de tipo que envolvem o construtor de tipo de funções ($\to$):

$$\tau_1 \to \tau_2 \equiv ((\to)\, \tau_1)\, \tau_2$$

e a notação $[\tau]$ será usada para expressar a aplicação do construtor de listas (que possui $\star \to \star$) ao tipo τ:

$$[\tau] \equiv ([\,])\, \tau$$

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

$$\mathrm{tv}(\tau) = \begin{cases} \{\alpha\} & \text{se } \tau = \alpha \\ \emptyset & \text{se } \tau = C \\ V_1 \cup V_2 & \text{se } \tau = \tau_1\tau_2,\ V_1 = \mathrm{tv}(\tau_1) \text{ e } V_2 = \mathrm{tv}(\tau_2) \end{cases}$$

Também usamos a expressão $\mathrm{tv}(X)$ para denotar a família de variáveis de um conjunto $X \subseteq \mathbf{T}$ de expressões de tipo:

$$\mathrm{tv}(X) = \bigcup \{\mathrm{tv}(\tau) \mid \tau \in X\}$$

Restrições

Expressões com a forma $x : \tau$ denotam *restrições*, representadas também por variáveis com a forma c, c', etc.

A variável κ e expressões com a forma $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$, para todo $n \geq 0$, denotam *conjuntos de restrições.*

O conjunto de variáveis de tipo em uma restrição é dado pela função $\mathrm{tv} : \mathbf{K} \to \mathcal{P}(\mathbf{V})$, onde $\mathrm{tv}(x : \tau) = \mathrm{tv}(\tau)$. O conjunto de variáveis é a união dos conjuntos de variáveis de tipo de cada restrição:

$$\mathrm{tv}(\kappa) \quad = \quad \bigcup_{1 \leq i \leq n} \mathrm{tv}(\tau_i) \text{ para } \kappa = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}, \text{ e } n \geq 0$$

Tipos

Uma expressão $\forall \bar{\alpha}.\, \kappa \,.\, \tau$ denota um *tipo*. Se $\kappa \neq \emptyset$, dizemos que é um *tipo restringido*, caso contrário, dizemos que é um *tipo irrestrito*. Se $\bar{\alpha} = \emptyset$, dizemos que é um *tipo monomórfico*, caso contrário será um tipo polimórfico. Para simplificar a sintaxe, usamos as seguintes abreviações:

- $\forall \bar{\alpha}.\tau$ quando $\kappa = \emptyset$;
- $\kappa \,.\, \tau$ quando $\bar{\alpha} = \emptyset$;
- τ quando $\kappa = \bar{\alpha} = \emptyset$.

O conjunto de variáveis livres de um tipo σ é dado pela função $\mathrm{tv} : \mathbf{J} \to \mathcal{P}(\mathbf{V})$, definida como:

$$\mathrm{tv}(\forall \bar{\alpha}.\kappa.\tau) = (\mathrm{tv}(\kappa) \cup \mathrm{tv}(\tau)) - \bar{\alpha}$$

Como no caso de tipos simples, também usamos a notação $\mathrm{tv}(X)$ para denotar o conjunto de variáveis livres de um conjunto de tipos:

$$\mathrm{tv}(X) = \bigcup\{\mathrm{tv}(\sigma) \mid \sigma \in X\}, \text{ para } X \subseteq \mathbf{J}$$

4.3 Substituições

Uma substituição S é uma função de variáveis de tipo para expressões de tipo simples. O conjunto de todas as substituições é denotado por $\mathbf{S}$ e a substituição identidade é denotada por id.

Escrevemos $\mathrm{dom}(S)$ para representar o conjunto de variáveis para as quais a imagem sob S é diferente da imagem da substituição identidade:

$$\mathrm{dom}(S) = \{\alpha \mid S(\alpha) \neq \alpha\}$$

Através deste documento, consideraremos apenas substituições para as quais o conjunto $\mathrm{dom}(S)$ é finito (substituições finitas), e que preservam o *kind* das variáveis, i.e. $S(\alpha^k) \in \mathbf{T}^k$ para qualquer *kind* k.

Toda substituição S pode ser estendida para um homomorfismo sobre expressões de tipo simples usando o operador $(_)^*$, definido como:

$$\begin{aligned} S^*(\alpha) &= S(\alpha) \\ S^*(\mathrm{C}) &= \mathrm{C} \\ S^*(\tau_1\tau_2) &= S^*(\tau_1)\, S^*(\tau_2) \end{aligned}$$

Por razões de simplicidade, escreveremos $S(\tau)$ ao invés de $S^*(\tau)$, e $S\tau$ ao invés de $S(\tau)$.

O termo $[\alpha \mapsto \tau]$ denota a substituição definida como:

$$[\alpha \mapsto \tau](\beta) = \begin{cases} \tau & \text{se } \alpha = \beta \\ \beta & \text{se } \alpha \neq \beta \end{cases}$$

Substituições podem ser estendidas para homomorfismos sobre restrições ($S\delta$), conjuntos de restrições ($S\kappa$) e tipos restringidos ($S(\kappa.\tau)$), i.e. aplicando a substituição a cada variável de tipo que aparece como subtermo. A extensão de uma substituição para uma função de conjuntos de tipos para conjuntos de tipos é definida como:

$$S(X) = \bigcup\{S\tau \mid \tau \in X\}$$

onde X é um conjunto de tipos.

Uma substituição também pode ser estendida para uma operação sobre tipos polimórficos, mas é preciso ter cuidado para evitar a captura de variáveis ligadas. A aplicação de uma substituição S a um tipo σ é dita *livre de captura* se $\mathrm{tv}(S\sigma) = \mathrm{tv}(S(\mathrm{tv}(\sigma))$.

A função tv também pode ser estendida para substituições, conforme a definição abaixo:

$$\mathrm{tv}(S) = \mathrm{dom}(S) \cup \{\mathrm{tv}(S(\alpha)) \mid \alpha \in \mathrm{dom}(S)\}$$

Uma substituição pode ser especificada por um conjunto de pares $(\alpha \mapsto \tau)$. Para $n \geq 1$ e para $\alpha_i \neq \alpha_j$, com $1 \leq i < j \leq n$, a notação $[\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$ denota a substituição S tal que:

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

4.4 Visão geral do Sistema CT

Recursão polimórfica

O Sistema CT, como outros sistemas de tipos baseados no sistema de Damas-Milner, não permite a abstração polimórfica: parâmetros de funções não podem assumir tipos polimórficos. No entanto, por ser baseado também no sistema de Milner-Mycroft [30], o Sistema CT permite a *recursão polimórfica* [36]. Em outras palavras, o Sistema CT permite que ocorrências recursivas de uma função assumam, no corpo da função, tipos distintos não unificáveis.

A inferência de tipos na presença de recursão polimórfica foi mostrada ser equivalente ao problema de semi-unificação [14, 12], que por sua vez foi demonstrado ser indecidível [21]. No entanto, existem semi-algoritmos para o problema [12, 36] que terminam em todos os casos em que uma solução existe e um tipo válido pode ser inferido para uma função polimórfico-recursiva e em uma ampla classe de casos em que soluções não existem (em que um tipo válido não pode ser atribuído a uma função polimórfico-recursiva).

Contextos de tipagem

Um *contexto de tipagem* $\Gamma \subseteq \mathbf{X} \times \mathbf{J}$ é uma relação entre variáveis de termo e tipos. Como tal, ele pode ser visto como um conjunto de *suposições de tipo* que tem a forma $x : \sigma$, onde x é uma variável de termo (ou símbolo) e σ é um tipo. Um contexto de tipagem representa uma informação sensível a contexto sobre um contexto do programa, e é usada na definição de regras de tipagem para representar as *suposições de tipo* de símbolos usados em um determinado contexto do programa.

O conjunto $\Gamma(x) = \{\sigma \mid x : \sigma \in \Gamma\}$ representa o conjunto de tipos associados ao símbolo x no contexto Γ.

Como o Sistema CT permite a sobrecarga de símbolos, um contexto de tipagem pode incluir mais de uma suposição para um dado símbolo. Um símbolo x é sobrecarregado em um contexto de programa se tiver mais de uma suposição correspondente no contexto de tipagem, i.e. $|\Gamma(x)| \geq 2$.

Política de sobrecarga

Nem todos os subconjuntos de $\mathbf{X} \times \mathbf{J}$ representam contextos válidos, mas apenas aqueles que satisfazem a política de sobrecarga do Sistema CT, que diz que tipos de suposições

distintas do mesmo símbolo em um contexto não podem se sobrepor. Tipos $\forall\bar{\alpha}.\,\kappa.\,\tau$ e $\forall\bar{\alpha}'.\kappa'.\,\tau'$, com $(\bar{\alpha} \cup \bar{\alpha}') \cap (\mathrm{tv}(\tau) \cup \mathrm{tv}(\tau')) = \emptyset$ se sobrepõem se τ e τ' são unificáveis, i.e. existe uma substituição S tal que $S\tau = S\tau'$ com $\mathrm{dom}(S) \subseteq \bar{\alpha} \cup \bar{\alpha}'$.

Para um tratamento mais detalhado sobre políticas de sobrecarga, veja [6].

4.5 Semi-reticulados

Esta seção define ordens parciais e seus semi-reticulados correspondentes [5] para o conjunto de expressões de tipo simples e para o conjunto de substituições. Também são apresentadas operações relacionadas e funções que serão usadas na definição da satisfazibilidade de conjuntos de restrições.

Expressões de tipos simples

Pré-ordem de expressões tipos simples

A relação $\preceq_{\mathrm{T}}$ é uma pré-ordem sobre o conjunto de expressões de tipo simples, definida como:

$$\tau \preceq_{\mathrm{T}} \tau' \text{ se e somente se } S\tau = S\tau', \text{ para algum } S$$

Se $\tau \preceq_{\mathrm{T}} \tau'$ é verdade, e $\bar{\alpha} = \mathrm{tv}(\tau)$ e $\bar{\alpha}' = \mathrm{tv}(\tau')$, dizemos que $\forall\bar{\alpha}'.\tau'$ é mais geral do que $\forall\bar{\alpha}.\tau$, ou que é uma generalização de $\forall\bar{\alpha}.\tau$.

Equivalência de expressões de tipo simples módulo renomeamento de variáveis

Usando a pré-ordem $\preceq_{\mathrm{T}}$, podemos definir a relação de equivalência $\equiv_{\mathrm{T}}$ como:

$$\tau \equiv_{\mathrm{T}} \tau' \text{ se e somente se } \tau \preceq_{\mathrm{T}} \tau' \text{ e } \tau' \preceq_{\mathrm{T}} \tau$$

Se $\tau \equiv_{\mathrm{T}} \tau'$ é verdade, dizemos que τ é equivalente a τ' módulo renomeamento de variáveis.

Ordem parcial de expressões de tipo simples

Tomando $\mathrm{T}_{\equiv}$ como o conjunto das classes de equivalência de $\equiv_{\mathrm{T}}$, podemos estender a pré-ordem $\preceq_{\mathrm{T}}$ para uma ordem parcial sobre $\mathrm{T}_{\equiv}$. Usando a notação $[\tau]_{\equiv}$ para representar a classe de equivalência da expressão de tipo simples τ módulo $\equiv_{\mathrm{T}}$ ($[\tau]_{\equiv} = \{\tau' | \tau' \equiv_{\mathrm{T}} \tau\}$) definimos a ordem parcial $\leq_{\mathrm{T}}$ como:

$$[\tau]_{\equiv} \leq_{\mathrm{T}} [\tau']_{\equiv} \text{ se e somente se } \tau \preceq_{\mathrm{T}} \tau'$$

O elemento $[\alpha]$, onde α é qualquer variável de tipo, possui a propriedade de ser o *majorante* da ordem parcial, ou seja, para todo τ, $[\tau] \leq_{\equiv} [\alpha]$.

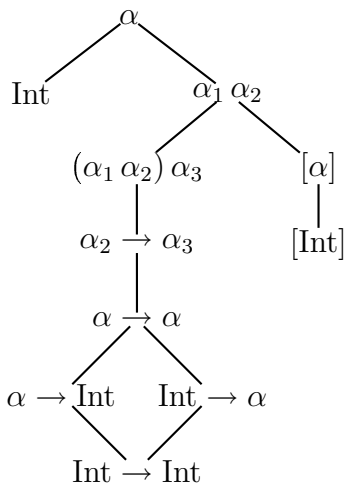

Figura 4.1: Um fragmento do semi-reticulado de tipos simples

Como qualquer elemento de uma classe de equivalência pode ser usado para representar a classe, omitiremos os colchetes e o sinal $\equiv$ quando nos referirmos a elementos de $\mathrm{T}_{\equiv}$, escrevendo τ ao invés de $[\tau]_{\equiv}$.

Semi-reticulado de expressões de tipo simples

Dizemos que $[\tau] = [\tau_1] \vee [\tau_2]$, ou simplesmente que $\tau = \tau_1 \vee \tau_2$ (τ é o supremo de τ_1 e τ_2), se τ for mais geral que τ_1 e τ_2, e se τ for o elemento menos geral com esta propriedade. A prova de que todos os pares de elementos de $\mathrm{T}_{\equiv}$ possuem supremos será omitida, pois está além do escopo deste documento. Um algoritmo para calcular $\vee$ será descrito na próxima seção.

O supremo de um conjunto finito não-vazio $X \subseteq \mathrm{T}_{\equiv}$ ($\bigvee X$), é definido recursivamente como:

$$\bigvee X = \begin{cases} \tau & \text{se } X = \{\tau\} \\ \tau \vee \tau' & \text{se } X = \{\tau\} \cup X' \text{ e } \tau' = \bigvee X' \end{cases}$$

A tupla ($\mathrm{T}_{\equiv}$,$\bigvee$,$[\alpha]$) forma um semi-reticulado, com a propriedade que todos os conjuntos não-vazios e finitos de elementos de $\mathrm{T}_{\equiv}$ têm um supremo. Novamente, a prova desta última afirmação será omitida por estar além do escopo deste documento.

A figura 4.1 descreve um fragmento do semi-reticulados de expressões de tipo simples. Note que o tipo simples $\alpha \to \alpha$ é o supremo de Int $\to \alpha$ e $\alpha \to$ Int, enquanto que o tipo simples $\alpha_1\ \alpha_2$ é o supremo dos tipos simples Int $\to$ Int e [Int].

Condição de cadeia crescente finita

Uma cadeia crescente é uma sequência $\{a_1, a_2, \ldots, a_n\}$ de elementos de um conjunto parcialmente ordenado $(P, \leq)$ tais que $a_1 < a_2 < \cdots < a_n$. O conjunto parcialmente ordenado $(\mathrm{T}_{\equiv}, \leq_{\mathrm{T}})$ tem a propriedade de que todas as cadeias crescentes são finitas. Uma destas cadeias é a sequência $\mathrm{Int} \to \mathrm{Int} < \alpha \to \mathrm{Int} < \alpha \to \alpha < \alpha_1 \to \alpha_2 < (\alpha_1\,\alpha_2)\,\alpha_3 < \alpha_1\,\alpha_2 < \alpha$ mostrada na figura 4.1.

A prova de que todas as cadeias crescentes de $\leq_{\mathrm{T}}$ são finitas será omitida.

Substituições

Pré-ordem das substituições

A pré-ordem $\preceq_{\mathrm{S}}$ sobre o conjunto de substituições é definida como:

$$S_1 \preceq_{\mathrm{S}} S_2 \text{ se, e somente se, existe um } S \text{ tal que } S \circ S_2 = S_1$$

A pré-ordem $\preceq_{\mathrm{S}}$ induz uma relação de equivalência, $\equiv_{\mathrm{S}}$, definida como:

$$S_1 \equiv_{\mathrm{S}} S_2 \text{ se, e somente se, } S_1 \preceq_{\mathrm{S}} S_2 \text{ e } S_2 \preceq_{\mathrm{S}} S_1$$

Ordem parcial de substituições

Tomando $\mathrm{S}_{\equiv}$ como o conjunto de classes de equivalência de substituições módulo $\equiv_{\mathrm{S}}$, e $[S]_{\equiv}$ como a classe de equivalência da substituição S, podemos definir a ordem parcial $\leq_{\mathrm{S}}$ sobre $\mathrm{S}_{\equiv}$ da seguinte forma:

$$[S]_{\equiv} \leq_{\mathrm{S}} [S']_{\equiv} \text{ se, e somente se, } S \preceq_{\mathrm{S}} S'$$

Se $S \preceq_{\mathrm{S}} S'$ é verdade, dizemos que S' é mais geral que S (ou uma generalização de S), ou ainda que S é mais específico que S' (ou uma especialização de S').

A substituição identidade (id) é o *majorante* desta ordem parcial.

Novamente, omitiremos os colchetes e o símbolo $\equiv$ quando nos referirmos a elementos de $\mathrm{S}_{\equiv}$, escrevendo S ao invés de $[S]_{\equiv}$.

Semi-reticulado de substituições

Da mesma maneira que definimos o semi-reticulado dos tipos simples, podemos definir o semi-reticulado de substituições $(\mathrm{S}_{\equiv}, \bigvee, \mathrm{id})$, que também possui supremos de todos os conjuntos finitos não-vazios.

O algoritmo que calcula o supremo de duas substituições será mostrado na próxima seção.

Operações auxiliares

As funções abaixo, usadas neste capítulo, se aplicam a quaisquer conjuntos parcialmente ordenados.

Subconjunto mínimo e máximo

O subconjunto mínimo de um conjunto X de elementos de um conjunto parcialmente ordenado $(\mathrm{X}, \leq_\mathrm{X})$ é o conjunto de elementos mínimos de X, definido como:

$$\mathrm{minset}(X) = \{x \in X \mid \text{ para todo } x' \in X,\ (x \leq_\mathrm{X} x' \text{ ou } x' \not\leq_\mathrm{X} x)\}$$

Do mesmo modo, temos o subconjunto máximo de X, definido como:

$$\mathrm{maxset}(X) = \{x \in X \mid \text{ para todo } x' \in X,\ (x' \leq_\mathrm{X} x \text{ ou } x \not\leq_\mathrm{X} x')\}$$

Limite inferior e limite superior

Dado um conjunto X de elementos de um conjunto parcialmente ordenado $(\mathrm{X}, \leq_\mathrm{X})$, em um elemento x desta ordem, dizemos que o limite superior de x em X é o conjunto de elementos de X que são maiores ou iguais a x:

$$\mathrm{ub}(x, X) = \{x' \in X \mid x \leq x'\}$$

O limite inferior é definido de modo semelhante:

$$\mathrm{lb}(x, X) = \{x' \in X \mid x' \leq x\}$$

Cobertura superior e inferior

A cobertura superior de um elemento x de um conjunto parcialmente ordenado $(\mathrm{X}, \leq_\mathrm{X})$ em um conjunto $X \subseteq \mathrm{X}$ é o subconjunto mínimo do limite superior de x em X. Formalmente:

$$\mathrm{upcover}(x, X) = \mathrm{minset}(\mathrm{ub}(x, X))$$

A cobertura inferior é definida de maneira semelhante:

$$\mathrm{lwcover}(x, X) = \mathrm{maxset}(\mathrm{lb}(x, X))$$

4.6 Operações de supremo

Uma função que calcula o supremo de dois tipos simples é chamada lcg (menor generalização comum), e é definida como:

$$
\begin{aligned}
\mathrm{lcg}(\tau_1, \tau_2) &= \tau \quad \text{onde } (\tau, S_1, S_2, V) = \mathrm{lcg}'(\tau_1, \tau_2, \mathrm{id}, \mathrm{id}, \mathrm{tv}(\tau) \cup \mathrm{tv}(\tau')) \\
\mathrm{lcg}'(\mathrm{C}, \tau, S_1, S_2, V) &= \begin{cases} (\mathrm{C}, S_1, S_2, V) & \text{se } \tau = \mathrm{C} \\ \mathrm{gen}(\mathrm{C}, \tau, S_1, S_2, V) & \text{caso contrário} \end{cases} \\
\mathrm{lcg}'(\tau_1\,\tau_2, \tau_1'\,\tau_2', \mathrm{S_1}, \mathrm{S_2}, \mathrm{V}) &= \begin{cases} (\tau\,\tau', S, S', V_2) \\ \qquad \text{se } \mathrm{kind_T}(\tau_1) = \mathrm{kind_T}(\tau_1'), \\ \qquad\quad (\tau, S_1', S_2', V_1) = \mathrm{lcg}'(\tau_1, \tau_1', S_1, S_2, V), \\ \qquad\quad (\tau', S, S', V_2) = \mathrm{lcg}'(\tau_2, \tau_2', S_1', S_2', V_1) \\ \\ \mathrm{gen}(\tau_1\,\tau_2, \tau_1'\,\tau_2', S_1, S_2, V), \text{ caso contrário} \end{cases}
\end{aligned}
$$

em todos os outros casos:

$$
\mathrm{lcg}'(\tau_1, \tau_2, S_1, S_2, V) = \mathrm{gen}(\tau_1, \tau_2, S_1, S_2, V)
$$

A função gen é definida como:

$$
\mathrm{gen}(\tau_1, \tau_2, S_1, S_2, V) = \begin{cases} (\alpha, S_1, S_2, V), \text{ if } S_1(\alpha) = \tau_1, S_2(\alpha) = \tau_2, \text{ para algum } \alpha \\ (\alpha', S_1' \circ S_1, S_2 \circ S_2, V \cup \{\alpha'\}), \text{ caso contrário,} \\ \qquad \text{onde } \alpha' \notin V, \text{ e} \\ \qquad\quad S_1' = [\alpha' \mapsto \tau_1], S_2' = [\alpha' \mapsto \tau_2] \end{cases}
$$

A função gen é usada para definir a variável que representará, no tipo resultado da operação, dois subtermos correspondentes e não-unificáveis dos tipos a partir dos quais se está computando o lcg. Ela garante que uma mesma variável seja usada no lcg para representar os mesmos subtermos correspondentes[1]. A variável V representa o conjunto de variáveis ainda não usadas.

O supremo de duas substituições é computável pela função lcgs, definida como:

$$
\begin{aligned}
\mathrm{lcgs}(S_1, S_2) &= S, \text{ onde} \\
V &= \mathrm{tv}(S_1) \cup \mathrm{tv}(S_2) \\
(S, S', S'') &= \mathrm{lcgs}'(\mathrm{dom}(S_1) \cup \mathrm{dom}(S_2), \mathrm{id}, \mathrm{id}, \mathrm{id}) \\
\mathrm{lcgs}'(\emptyset, S, S_A, S_B) &= S \\
\mathrm{lcgs}'(\alpha, S_A', S_B', V') &= \mathrm{lcg}'(S_1(\alpha), S_2(\alpha), S_A, S_B, V) \\
\mathrm{lcgs}'(\{\alpha\} \cup V, S, S_A, S_B) &= \mathrm{lcgs}'(V, [\alpha \mapsto \tau] \circ S, S_A', S_B') \quad \text{se } V \neq \emptyset
\end{aligned}
$$

[1]Por exemplo, lcg(Int → Int, Float → Float) = $\alpha \to \alpha$ para alguma variável α, e não $\alpha \to \beta$, para variáveis distintas α e β.

4.7 Satisfazibilidade de conjuntos de restrições

Um conjunto de restrições κ em um tipo σ restringe o conjunto de tipos para o qual σ pode ser instanciado em um dado contexto Γ, conforme existam suposições em Γ que satisfaçam κ. A relação de satisfazibilidade será definida em alguns passos nesta seção, primeiramente para uma única restrição e depois para um conjunto de restrições.

Satisfazibilidade de uma restrição

A restrição $x : \tau$ representa um requerimento sobre o contexto. Sua presença no tipo de uma expressão indica que a expressão exige que uma definição para x com um tipo instanciável para τ esteja presente no contexto. $x : \tau$ é dito *satisfazível em um contexto* Γ se e somente se existe uma substituição S e uma instância de x em Γ cujo tipo possa ser instanciado para $S\tau$. Dizemos então que S é uma *solução* para a restrição $x : \tau$ no contexto de tipagem Γ.

Um tipo $\sigma = \forall\bar{\alpha}.\kappa.\tau$ *pode ser instanciado* para o tipo simples $\tau' = S\tau$ em um contexto Γ se S for uma solução de κ em Γ e $\mathrm{dom}(S) \subseteq \bar{\alpha}$.

Por exemplo, considere as restrições $x_1 : \alpha \to \mathrm{Int}$, $x_2 : \alpha \to \alpha$, $x_3 : \alpha \to \alpha$, e o contexto Γ_1 definido como:

$$\begin{aligned}\Gamma_1 \ = \{ \ & x_1 : \mathrm{Int} \to \mathrm{Int},\ x_1 : \mathrm{Bool} \to \mathrm{Int}, \\ & x_2 : \mathrm{Int} \to \mathrm{Int},\ \mathrm{x}_2 : \mathrm{Int}, \\ & x_3 : \mathrm{Int} \to \mathrm{Bool}\}\end{aligned}$$

A primeira restrição, $x_1 : \alpha \to \mathrm{Int}$, tem duas soluções: $S^1_{1,} = \{\alpha \mapsto \mathrm{Int}\}$ e $S^1_2 = \{\alpha \mapsto \mathrm{Bool}\}$, enquanto que a segunda restrição, $x_2 : \alpha \to \alpha$, tem apenas uma solução: $S^2 = \{\alpha \mapsto \mathrm{Int}\}$. A última restrição, $x_3 : \alpha \to \alpha$, não tem solução. Estes exemplos ilustram as três possíveis situações de uma restrição em um contexto:

- uma restrição é *satisfazível* mas *não resolvida* em um contexto Γ se possui mais de uma solução em Γ;
- uma restrição é *satisfazível* e *resolvida* em um contexto Γ se possui apenas uma solução em Γ;
- uma restrição é *insatisfazível* em Γ se não houverem soluções para a mesma em Γ.

Uma propriedade importante de uma solução S de uma restrição $x : \tau$ é a de que ela pode ser usada para construir uma solução para qualquer restrição $x : \tau'$ tal que τ' seja mais geral que τ. Isto acontece porque $x : \tau$ é uma restrição mais forte do que $x : \tau'$. Considerando S' a substituição tal que $\tau = S'\tau'$, temos que $S \circ S'$ é uma solução para $x : \tau'$.

A regra principal que define a satisfazibilidade de uma restrição é a regra SAT_1, definida como:

$$\frac{S\tau = S\tau' \quad \mathrm{dom}(S) \subseteq (\bar{\alpha} \cup \mathrm{tv}(\tau)) - \mathrm{tv}(\Gamma) \quad \Gamma \models_{S'} S\kappa}{\Gamma \cup \{x \,:\, \forall \bar{\alpha}.\kappa.\tau'\} \models_{S' \circ S} \; x \,:\, \tau} \quad (\mathrm{SAT}_1)$$

As primeiras duas condições, $S(\tau) = S(\tau')$ e $\mathrm{dom}(S) \subseteq (\bar{\alpha} \cup \mathrm{tv}(\tau)) - \mathrm{tv}(\Gamma)$, expressam que é preciso existir no contexto uma suposição para x cujo tipo seja unificável com o tipo especificado na restrição. A terceira condição, $\Gamma \models_{S'} S\kappa$, exige que o conjunto de restrições associadas à suposição, instanciado por S, seja também satisfazível no contexto.

Satisfazibilidade de um conjunto de restrições

A satisfazibilidade de um conjunto de restrições $\kappa = \{x_1 \,:\, \tau_1, \ldots, x_n \,:\, \tau_n\}$ é definida de maneira semelhante: κ é dito *satisfazível* em um contexto Γ se existe uma substituição S tal que S seja uma solução para todos os elementos de κ. Portanto, o conjunto vazio de restrições é trivialmente satisfazível, tendo qualquer substituição como solução.

Como um exemplo, o conjunto de restrições $\kappa_1 = \{x_1 \,:\, \alpha \to \mathrm{Int},\ x_2 \,:\, \alpha \to \alpha\}$ tem apenas uma solução Γ: $S = \{\alpha \mapsto \mathrm{Int}\}$, enquanto que o conjunto de restrições $\kappa_2 = \{x_2 \,:\, \alpha \to \alpha,\ x_3 \,:\, \beta \to \alpha\}$ não possui soluções em Γ.

Pela definição acima, temos que uma solução S para $x_1 : \tau_1$ e para $\kappa = \{x_2 \,:\, \tau_2, \ldots, x_n \,:\, \tau_n\}$ é também uma solução para $\{x_1 \,:\, \tau_1\} \cup \kappa$. Esta propriedade nos leva à formulação indutiva da relação de satisfazibilidade de conjuntos de restrições, representadas pelas regras $\mathrm{SAT}_\emptyset$ e SAT_{muitos}:

$$\frac{}{\Gamma \models_S \emptyset} \quad (\mathrm{SAT}_\emptyset)$$

$$\frac{\Gamma \models_S \; x \,:\, \tau \qquad \Gamma \models_S \; \kappa}{\Gamma \models_S \; \{x \,:\, \tau\} \cup \kappa} \quad (\mathrm{SAT}_{muitos})$$

Exemplo: uma restrição simples

Tomando $\Gamma = \{x \,:\, \mathrm{Int} \to \mathrm{Int},\ x \,:\, \mathrm{Bool} \to \mathrm{Int},\ x \,:\, \mathrm{Bool} \to \mathrm{Bool}\}$ como o contexto, temos que as seguintes árvores de derivação mostram a satisfazibilidade da restrição $x : \alpha \to \mathrm{Int}$, onde $S_1 = \{\alpha \mapsto \mathrm{Int}\}$ e $S_2 = \{\alpha \mapsto \mathrm{Bool}\}$

$$\frac{S_1(\alpha \to \mathrm{Int}) = S_1(\mathrm{Int} \to \mathrm{Int}) \quad \mathrm{dom}(S) \subseteq \{\alpha\} \quad \dfrac{}{\Gamma \models_{S_1} \emptyset}(\mathrm{SAT}_\emptyset)}{\Gamma \cup \{x \,:\, \mathrm{Int} \to \mathrm{Int}\} \models_{S_1} \; x \,:\, \alpha \to \mathrm{Int}}(\mathrm{SAT}_1)$$

$$\frac{S_2(\alpha \to \mathrm{Int}) = S_2(\mathrm{Bool} \to \mathrm{Int}) \quad \mathrm{dom}(S) \subseteq \{\alpha\} \quad \dfrac{}{\Gamma \models_{S_2} \emptyset}(\mathrm{SAT}_\emptyset)}{\Gamma \cup \{x \,:\, \mathrm{Bool} \to \mathrm{Int}\} \models_{S_2} \; x \,:\, \alpha \to \mathrm{Int}}(\mathrm{SAT}_1)$$

Exemplo: um conjunto de restrições

Tomando $\Gamma = \{x_1 \ : \ \mathrm{I} \to \mathrm{I},\ x_1 \ : \ \mathrm{I} \to \mathrm{B},\ x_2 \ : \ \mathrm{B}\}$ como contexto, temos que a seguinte árvore de derivação mostra que $\Gamma \models_S \ \{x_1 \ : \ \alpha \to \beta,\ x_2 \ : \ \beta\}$, onde $S = \{\alpha \mapsto \mathrm{I},\ \beta \mapsto \mathrm{B}\}$ (os nomes das regras são abreviações dos índices das regras SAT_1, $\mathrm{SAT}_{\mathrm{muitos}}$ e $\mathrm{SAT}_\emptyset$):

$$\dfrac{\dfrac{S(\alpha \to \beta) = S(\mathrm{I} \to \mathrm{B}) \quad \dfrac{}{\Gamma \models_S \emptyset}(\emptyset)}{\Gamma \cup \{x_1 : \mathrm{I} \to \mathrm{B}\} \models_S \ x_1 : \alpha \to \beta}(1) \quad \dfrac{\dfrac{S(\beta) = S(\mathrm{B}) \quad \dfrac{}{\Gamma \models_S \emptyset}(\emptyset)}{\Gamma \cup \{x_2 : \mathrm{B}\} \models_S \ x_2 : \beta}(1) \quad \dfrac{}{\Gamma \models_S \emptyset}(\emptyset)}{\Gamma \models_S \{x_2 : \beta\}}(\mathrm{m})}{\Gamma \models_S \{x_1 : \alpha \to \beta,\ x_2 : \beta\}}(\mathrm{m})$$

Exemplo: um conjunto de restrições com prova em dois níveis

Seja $\Gamma = \{x_1 \ : \ \mathrm{Int} \to \mathrm{Int},\ \mathrm{x_1} \ : \ \mathrm{Int} \to \mathrm{Bool},\ \mathrm{x_2} \ : \ \forall\gamma.\{\mathrm{x_1} \ : \ \gamma \to \gamma\}.[\gamma] \to [\gamma]\}$. Este contexto distingue-se dos contextos dos exemplos anteriores em que uma das instâncias usadas para satisfazer uma restrição é um tipo restringido, ou seja, também possui restrições. A prova de que $S = \{\alpha \mapsto [\mathrm{Int}],\ \beta \mapsto [\mathrm{Int}], \gamma \mapsto \mathrm{Int}\}$ é uma solução para a restrição $x_2 : \alpha \to \beta$ em Γ é mostrada abaixo:

$$\dfrac{S([\gamma] \to [\gamma]) = S(\alpha \to \beta) \quad \dfrac{\dfrac{S(\mathrm{Int} \to \mathrm{Int}) = S(\gamma \to \gamma) \quad \dfrac{}{\Gamma \models_S \emptyset}(\emptyset)}{\Gamma \cup \{x_1 \ : \ \mathrm{Int} \to \mathrm{Int}\} \models_S \ x_1 \ : \ \gamma \to \gamma}(1) \quad \dfrac{}{\Gamma \models_S \emptyset}(\emptyset)}{\Gamma \models_S \ \{x_1 \ : \ \gamma \to \gamma\}}(\mathrm{m})}{\Gamma \cup \{x_2 \ : \ \forall\gamma.\{x_1 \ : \ \gamma \to \gamma\}.[\gamma] \to [\gamma]\}\} \models_S \ x_2 \ : \ \alpha \to \beta}(1)$$

4.8 Regras de satisfazibilidade

As regras de satisfazibilidade descritas separadamente acima são apresentadas juntas na definição abaixo:

$$\dfrac{S\tau = S\tau' \quad \mathrm{dom}(S) \subseteq (\bar{\alpha} \cup \mathrm{tv}(\tau)) - \mathrm{tv}(\Gamma) \quad \Gamma \models_{S'} S\kappa}{\Gamma \cup \{x \ : \ \forall\bar{\alpha}.\kappa.\tau'\} \models_{S' \circ S} \ x \ : \ \tau} \quad (\mathrm{SAT}_1)$$

$$\dfrac{}{\Gamma \models_S \emptyset} \quad (\mathrm{SAT}_\emptyset)$$

$$\dfrac{\Gamma \models_S \ x \ : \ \tau \qquad \Gamma \models_S \ \kappa}{\Gamma \models_S \ \{x \ : \ \tau\} \cup \kappa} \quad (\mathrm{SAT}_{many})$$

O problema de satisfazibilidade de um conjunto de restrições κ em um contexto Γ é resolvido em [6] através do cálculo da substituição principal de κ em Γ. S é considerada uma substituição principal de κ em Γ se S for uma das substituições mais específicas dentre as substituições que, quando aplicadas a κ, resultam em um conjunto de restrições com

$$\frac{|\Gamma(x)| \geq 2 \quad \tau = \bigvee\{\tau' \mid \forall\bar{\alpha}.\kappa.\tau' \in \Gamma(x)\} \quad \bar{\alpha} = \mathrm{tv}(\tau)}{\Gamma \vdash x : \forall\bar{\alpha}.\{x : \tau\}.\tau} \quad (\mathrm{VAR}_{\mathrm{muitos}})$$

$$\frac{\Gamma(x) = \{\sigma\}}{\Gamma \vdash x \ : \ \sigma} \quad (\mathrm{VAR}_1)$$

$$\frac{\Gamma \vdash x : \forall\bar{\alpha}.\kappa.\tau \quad \mathrm{dom}(S) \subseteq \bar{\alpha} \quad \Gamma \models_S \kappa}{\Gamma \vdash x : S\kappa.\, S\tau} \quad (\mathrm{INST})$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \kappa_1.\tau_1 \quad \sigma_1 = \forall\bar{\alpha}.\kappa_1.\tau_1 \quad \bar{\alpha} = \mathrm{tv}(\kappa_1.\tau_1) - \mathrm{tv}(\Gamma) \\ \Gamma^x \cup \{x : \sigma_1\} \vdash e_2 : \kappa_2.\tau_2\end{array}}{\Gamma \vdash \mathrm{let}\, x = e_1 \,\mathrm{in}\, e_2 : \kappa_2.\tau_2} \quad (\mathrm{LET})$$

$$\frac{\Gamma, \{u : \tau'\} \vdash e : \kappa.\tau}{\Gamma \vdash \lambda u.e : \kappa.\tau' \to \tau} \quad (\mathrm{ABS})$$

$$\frac{\Gamma \vdash e_1 : \kappa_1.\tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \kappa_2.\tau_2 \quad \Gamma \models_S (\kappa_1 \cup \kappa_2)}{\Gamma \vdash e_1 e_2 : S(\kappa_1 \cup \kappa_2.\tau_2)} \quad (\mathrm{APPL})$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \kappa_1.\tau_1 \quad \sigma_1 = \forall\bar{\alpha}.\kappa_1.\tau_1 \quad \bar{\alpha} = \mathrm{tv}(\kappa_1.\tau_1) - \mathrm{tv}(\Gamma) \\ \Gamma \cup \{x : \sigma_1\} \vdash p : \kappa_2.\tau_2\end{array}}{\Gamma \vdash \mathrm{leto}\, x = e_1 \,\mathrm{in}\, p \ : \ S(\kappa_2.\tau_2)} \quad (\mathrm{LETO})$$

Figura 4.2: Sistema de tipos do Sistema CT

o mesmo conjunto de soluções que κ'. Como o problema de satisfazibilidade é indecidível [35, 38], o problema do cálculo da substituição principal também o é. No entanto um algoritmo incompleto para o problema, descrito em [6], tem sido usado com sucesso na implementação de algoritmo de inferência para um dialeto de Haskell que usa os tipos restringidos do Sistema CT ao invés de classes de tipos. Outro algoritmo será apresentado na seção 4.11.

4.9 Sistema de tipos

O sistema de tipos do Sistema CT é definido na figura 4.2 e é baseado no sistema de Damas-Milner [8]. Nota-se o uso da relação de satisfazibilidade ($\models$) nas regras INST e APPL.

A operação de remoção de suposição, usada na regra LET, é definida como:

$$\Gamma^x = \{y : \sigma \mid y : \sigma \in \Gamma,\, x \neq y\}$$

4.10 Inferência de tipos

Um tratamento mais detalhado sobre a inferência de tipos para o Sistema CT está além do escopo deste trabalho. No entanto, apresentaremos aqui uma descrição hipotética de um algoritmo de inferência, na forma de uma sequência de passos, para que possamos mostrar mais à frente onde o algoritmo de cálculo da substituição principal de um conjunto de restrições de registro (capítulo 6) deverá se inserir.

O algoritmo é divido em cinco passos: inferência de tipagens principais; inferência de substituições principais de conjuntos de restrições; especialização de tipos restringidos; simplificação de conjuntos de restrições e generalização de variáveis de tipos.

O primeiro passo calcula a tipagem principal [15] de cada símbolo definido no escopo mais externo do programa. Cada tipagem é representada por uma atribuição de tipo restringido $x : \{x_1 : \tau_1, \ldots, x_n : \tau_n\}.\tau$, tal que: x é o símbolo definido no escopo superior; τ é o tipo simples inferido para esta definição de x; $x_1, \ldots, x_n$ são símbolos livres usados na definição; e $x_1 : \tau_1, \ldots, x_n : \tau_n$ são as restrições que representam as exigências sobre os símbolos livres para que x possa ter o tipo τ nesta definição. As tipagens principais são agrupadas em um contexto $\Gamma_{relaxed}$.

Este algoritmo é capaz de lidar com recursão polimórfica, ao reduzir o problema do cálculo da tipagem principal ao problema de semi-unificação [20, 12], que pode ser resolvido por um semi-algoritmo [37] que pára em todos os casos em que uma solução existe e em alguns casos em que soluções não existem. Para evitar a não terminação, um limite de iterações é usado para parar o algoritmo.

Se uma ou mais tipagens principais não puderem ser calculadas, devido a um erro de tipo ou ao alcance do limite de iterações, o processo de inferência é terminado, apresentando uma mensagem de erro ao programador.

O segundo passo acha, para cada tipo restringido $x : \kappa.\tau$, a substituição principal S^κ de κ contexto $\Gamma_{relaxed}$. Um algoritmo que calcula S^κ está definido em [6] e outro é apresentado na próxima seção.

O terceiro passo especializa as suposições restringidas de $\Gamma_{relaxed}$, aplicando a cada uma a substituição principal correspondente calculada na etapa anterior, resultando no contexto $\Gamma_{tight} = \{x : S^\kappa(\kappa.\tau) \mid x : \kappa.\tau \in \Gamma_{relaxed}\}$. Neste contexto, as suposições para cada símbolo não podem se sobrepor, o que significa que para todo x, x', κ, κ', τ, τ' e S, se $x : \kappa.\tau \in \Gamma_{tight}$, $x' : \kappa'.\tau' \in \Gamma_{tight}$ e $S\tau = S\tau'$ então $x \neq x'$.

O quarto passo simplifica os conjuntos de restrições que aparecem no contexto, removendo restrições resolvidas, que são desnecessárias, de Γ_{tight} . Formalmente, $\Gamma_{resolved} = \{x : \kappa'.\tau \mid x : \kappa.\tau \in \Gamma_{tight}$ e $\kappa' = \kappa - \{c \in \kappa \mid |\text{inst}(c, \Gamma_{tight})| = 1\}$, onde $\text{inst}(c, \Gamma_{tight})$ é o conjunto de instâncias que satisfazem a restrição c em κ.

O passo final é a generalização das variáveis das suposições de $\Gamma_{resolved}$: $\Gamma_{final} = \{x : \forall\bar{\alpha}.\kappa.\tau \mid x : \kappa.\tau \in \Gamma_{resolved}$ e $\bar{\alpha} = \text{fv}(\kappa.\tau)\}$.

O contexto final, Γ_{final}, representa o tipo principal das definições presentes no programa.

Os passos do algoritmo e os resultados intermediários são apresentados na figura 4.3.

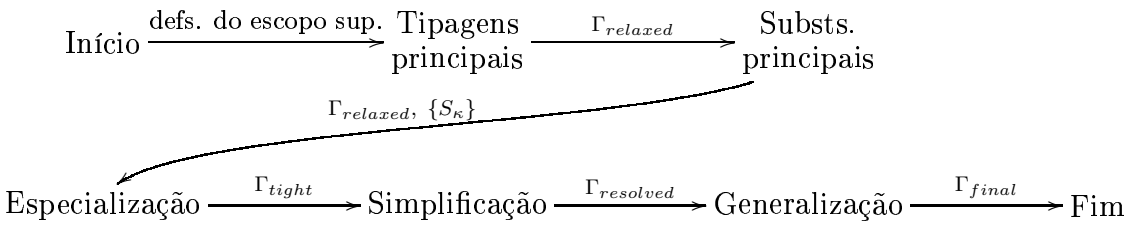

Figura 4.3: Visão geral do processo de inferência de tipos do Sistema CT.

4.11 Substituições principais

Como visto na seção anterior, um dos passos da inferência de tipos para o Sistema CT é o cálculo da substituição principal de conjuntos de restrições que aparecem no contexto de tipagem do programa. Neste passo ou descobrimos que um dos conjuntos de restrições que aparece no contexto é insatisfazível ou encontramos as substituições principais dos conjuntos de restrições presentes no contexto. A substituição principal é usada para calcular o tipo principal de cada definição em um programa.

Para calcular a substituição principal para um conjunto de restrições κ em um contexto Γ, precisamos achar um conjunto finito $\mathbb{S}$ de soluções de κ que represente todas as possíveis soluções de κ, i.e. se $\Gamma \models_S \kappa$ então deve existir uma substituição $S' \in \mathbb{S}$ e uma substituição S'' tais que $S\kappa = S''(S'\kappa)$.

O algoritmo que computa a substituição principal para um conjunto de restrições é apresentado na figura 4.4 sob a forma do julgamento determinístico '$\Gamma \Vdash S$ é principal para κ (S é uma substituição principal para κ em Γ).

O conjunto de soluções $\mathbb{S}$ de κ em Γ é construído considerando-se todas as possíveis formas de se resolver o conjunto de restrições κ usando-se as definições encontradas em Γ. Para cada restrição $x : \tau$ em κ, as instâncias de x em Γ que podem ser usadas para resolver $x : \tau$ são selecionadas pela função satset, que seleciona as instâncias de x cujos tipos pertencem ao conjunto candidates($x : \tau$, Γ).

Dada uma restrição $x : \tau$ e um conjunto de tipos $\mathbb{T} = \Gamma(x)$, candidates($x : \tau$, Γ) é definido como o conjunto dos tipos mais gerais da união de:

1. Subconjunto de $\mathbb{T}$ que cobre τ. Um conjunto $X \subseteq \mathbb{T}$ cobre τ se consistir dos elementos mínimos, ou tipos mais específicos, do conjunto de tipos de $\mathbb{T}$ que é igual ou mais geral que τ.

2. Subconjunto de $\mathbb{T}$ cujos tipos são mais específicos que τ.

As definições formais das funções satset e candidates, que fazem uso das funções definidas na seção 4.5, são:

$$\frac{\Gamma \Vdash_{\mathbb{S}} \kappa \quad \mathbb{S} \neq \emptyset \quad S = \bigvee \mathbb{S}}{\Gamma \Vdash S \text{ é principal para } \kappa} \qquad (\mathrm{SAT}_{princ})$$

$$\frac{\Gamma \Vdash_{\mathbb{S}_1} x : \tau \qquad \mathbb{S} = \{S_2 \circ S_1 \mid S_1 \in \mathbb{S}_1, \Gamma \Vdash_{\mathbb{S}_2} \kappa, S_2 \in \mathbb{S}_2\}}{\Gamma \Vdash_{\mathbb{S}} \{x : \tau\} \cup \kappa} \qquad (\mathrm{SAT}_{princ-\kappa})$$

$$\frac{}{\Gamma \Vdash_{\{\mathrm{id}\}} \emptyset} \qquad (\mathrm{SAT}_{princ-\emptyset})$$

$$\frac{\mathbb{S} = \{S_1 \cdot S \mid (S, \kappa) \in \mathrm{satset}(x : \tau, \Gamma), \quad \Gamma^{x,S\tau}, x : S\tau \Vdash_{\mathbb{S}_1} \kappa, \quad S_1 \in \mathbb{S}_1\}}{\Gamma \Vdash_{\mathbb{S}} x : \tau} \qquad (\mathrm{SAT}_{princ-1})$$

Figura 4.4: Regras de inferência para o cálculo da substituição principal.

$$\begin{aligned} \Gamma(x) &= \{\tau \mid x : \kappa.\tau \in \Gamma\} \\ \mathrm{candidates}(x : \tau, \Gamma) &= \mathrm{maxset}\,(\mathrm{upcover}\,(\tau, \Gamma(x)) \cup \mathrm{lb}\,(\tau, \Gamma(x))) \\ \mathrm{satset}(x : \tau, \Gamma) &= \{(S, S\kappa) \mid x : \kappa.\tau' \in \Gamma, \tau' \in \mathrm{candidates}(x : \tau, \Gamma), S\tau = S\tau'\} \end{aligned}$$

Para evitar a não-terminação em alguns casos, uma instância irrestrita de x é introduzida no contexto quando é feita a prova da satisfazibilidade do conjunto de restrições que aparece no tipo restringido de uma instância de x. Antes de inserir esta instância de x no contexto, removemos primeiro todas as instâncias de x que têm o mesmo tipo que ela ($\Gamma^{x,\tau}$):

$$\Gamma^{x,\tau} = \{x' : \kappa.\tau' \mid x' \neq x \text{ ou } \tau' \leq \tau\}$$

Capítulo 5

Acrescentando polimorfismo de registro ao Sistema CT

5.1 Objetivos do projeto

A proposta descrita neste documento procura definir um sistema de registros que possua as seguintes características:

- valores de registros devem ser *leves*: não deve ser necessária a declaração prévia de um tipo registro para que os valores correspondentes ao mesmo possam ser usados no programa.
- casamento de padrões polimórfico: o casamento de padrão sobre campos de registros deve ser feito sem que seja necessário o conhecimento em tempo de compilação do tipo do registro, como ocorre em Haskell ou Standard ML.
- manipulação polimórfica de registros: a atualização, extensão e restrição de registros devem ser polimórficas. Por exemplo, a operação de restrição sobre o campo $\mathtt{f}_1$ deve ser aplicável à qualquer registro que possua o campo $\mathtt{f}_1$.
- devem existir funções correspondentes a todas as operações de registro.
- os tipos dados à operações polimórficas de registros devem refletir, de uma forma intuitiva as exigências impostas às variáveis de tipo que ocorrem nos tipos dos parâmetros e no resultado das mesmas.
- o sistema de registro deve ser acrescentado ao Sistema CT como uma extensão.

5.2 Operações de registros

Antes de apresentar o sistema de tipos, definiremos primeiro as operações de registro que serão permitidas.

Construção de valores de registro

Registros são construídos em Haskell acrescentando-se uma lista de atribuições de campos a um construtor de dados associado a um tipo registro. Por exemplo:

```
data ... = RecCons { field1 :: ···, field2 :: ···, field3 :: ··· }

recValue = RecCons { field1 = ···, field2 = ···, field3 = ··· }
```

Ao contrário de Haskell, registros no sistema proposto são *leves*, portanto não é necessário especificar um construtor de dados na construção de valores de tipo registro. Desta forma, o valor acima pode ser especificado como:

```
recValue = { field1 = ···, field2 = ···, field3 = ··· }
```

Casamento de padrões de valores de tipo registro

Haskell permite o casamento de padrões sobre campos de um registro. A sintaxe é semelhante à sintaxe de construção de um registro, porém não é necessária a inclusão de todos os campos do registro correspondente. Por exemplo:

`recFunction (RecCons {field1 =` pat_1`, field2 =` pat_2`}) = ···`

Assim como na construção de valores, o sistema proposto também não exige que se especifique um construtor de dados. No entanto, devemos considerar duas hipóteses para o casamento de padrões:

- O programador especifica o conjunto completo de campos que um valor de registro deve ter:

  `recFunction {field1 =` pat_1`, field2 =` pat_2`} = ···`

  Neste caso, a função `recFunction` só aceitará registros cujos campos são `field1` e `field2`.

- O programador especifica parte do conjunto de campos que o valor de registro deve ter:

  `recFunction {field1 =` pat_1`, field2 =` pat_2`, .. } = ···`

  Neste caso, a função `recFunction` aceitará quaisquer valores de tipo registro que possuam pelo menos os campos `field1` e `field2`.

Outra variação conveniente, presente no sistema proposto, é o uso de variáveis de campos implícitas (*prunned fields*). De modo semelhante à linguagem Standard ML, se um campo for especificado em um casamento de padrão sem que um padrão seja especificado para o seu valor, presume-se que o casamento de padrão do valor do campo é feito sobre uma variável com o mesmo nome do campo. Por exemplo, a expressão

```
fieldPlus1 {field1, ..} = field1 + 1
```

é equivalente à:

```
fieldPlus {field1 = field1, ..}  = field1 + 1
```

Atualização de registros

A sintaxe proposta acima para a construção de registros conflita com a sintaxe de Haskell. Por exemplo, a seguinte função usa a sintaxe de atualização de registros de Haskell:

```
updFunc r = r { field1 = ···, field2 = ··· }
```

Com a sintaxe proposta, a expressão `r {field1 = , field2 = ···}` também pode ser interpretada como a passagem do valor de registro `{field1 = ···, field2 = ···}` para a função denotada por `r`.

Uma sintaxe alternativa para a atualização de registros, que será adotada nesta proposta, é descrita em [19]:

```
updFunc r = { r | field1 ← ···, field2 ← ··· }
```

Esta alternativa também apresenta um problema: parece-se com a sintaxe de listas.

Extensão e restrição de registros

Com algumas modificações, a sintaxe descrita acima pode ser estendida para fazer outros tipos de manipulações, como mostra o exemplo a seguir:estende

- O campo `field1`, com valor `v`, acrescentado a um registro:
  `{ r | ···, +field1 ← v, ··· }`.
- Um registro com o campo `field1` removido: `{ r | ···, -field1, ··· }`.

A seguinte função mostra outro exemplo:

```
recFun r@{ field1, ..} = { r | field1 ← field1 + 1,
                               +field2 ← 10,   -field3}
```

Esta função aceita como parâmetro qualquer valor de registro que possua os campos `field1` e `field3`, e que não possua o campo `field2`, e retorna o mesmo registro com o valor do campo `field1` acrescido de 1, com o valor do campo `field2` igual a 10 e sem o campo `field3`.

Esta sintaxe tem uma restrição: em uma expressão de manipulação de registros `{r|...}`, duas operações não podem usar o mesmo nome de campo. Por exemplo, as seguintes expressões são inválidas:

```
{ r | password <- "pass", nounce = 100, -password }
{ r | password <- "pass", password = "pass2" }
{ r | +password <- "pass", password = "pass2" }
```

A razão para esta restrição é que o uso de duas ou mais operações sobre o mesmo campo em uma mesma expressão de manipulação de registro será:

- ambígua, porque o resultado dependerá da ordem em que as operações serão feitas; ou
- redundante: se uma ordem de execução é escolhida, somente a última operação será significante, para cada campo.

O valor do registro vazio

Se permitirmos valores *leves* de registro, devemos considerar como representar o valor de um registro que não possui campos. Algumas propostas sugerem o uso do valor '`()`', que representa o valor de tipo *unit* em Haskell, para representar o registro vazio. Nesta proposta usaremos a sintaxe '`{}`' para representar o registro vazio.

Usando '`{}`', podemos definir a construção de registro como uma abreviação. Por exemplo, a expressão

```
{field1 = v1, field2 = v2}
```

pode ser vista como uma abreviação para:

```
{ {} | +field1 = v1, +field2 = v2 }.
```

Funções correspondentes às operações de registro

O sistema de registros de Haskell disponibiliza para o programador uma função de projeção para cada campo de um registro, onde o nome da função é o nome do campo. Se o tipo do campo é τ, e se o campo foi declarado em um construtor de dados para o tipo de dados `T`, então o tipo da função de projeção é $\texttt{T} \rightarrow \tau$.

De maneira semelhante, uma função de atualização de registro poderia ser automaticamente criada para cada campo. Porém, a escolha do nome da função não é óbvia: poderia

↓	↑	Δ	∩
≤	≥	⌂	≡

Figura 5.1: Alguns símbolos Unicode.

ser `fUpdate`, `update_f` ou `set_f`, por exemplo. Qualquer escolha teria como consequência uma poluição ainda maior do espaço de nomes de funções.

A linguagem Standard ML usa o símbolo `#` para distinguir o espaço de nomes de funções do espaço de nome das funções de projeção: a expressão `#f` denota a projeção sobre o campo `f`. Com isso, as funções de projeção não poluem o espaço de nomes das funções definidas pelo programador.

Usaremos esta idéia no sistema de registros proposto para separar o espaço de nomes das funções de manipulação de registros das funções definidas pelo programador:

- `#nomedocampo`: função de projeção para o campo `nomedocampo`, definida como:

```
#nomedocampo {nomedocampo, ..} = nomedocampo
```

- `#+nomedocampo`: função de extensão para o campo `nomedocampo`, definida como:

```
#+nomedocampo v r = { r | +nomedocampo ← v }
```

- `#-nomedocampo`: função de restrição para o campo nomedocampo, definida como:

```
#-nomedocampo r = { r | -nomedocampo }
```

Não haverá uma função específica para a atualização de registros. Porém, a função de atualização pode ser construída através da composição das funções de restrição e de extensão:

```
{ r | field ← v } = #+field v . #-field
```

Apesar desta solução evitar a poluição do espaço de nomes de funções, ela reduz o espaço de nomes de símbolos: os símbolos `#`, `#-` e `#+` devem ser reservados. Porém, ainda há muitos símbolos disponíveis usando codificação Unicode. A figura 5.1 mostra alguns dos símbolos disponíveis.

Se considerarmos as funções de manipulação de registros como operações primitivas, podemos definir o significado de uma expressão de casamento de padrão de campos de registro e de uma expressão de manipulação de registros através das regras de reescrita $\gg_P$ e $\gg_M$:

$$
\begin{array}{rcl}
\texttt{case}\ e_r\ \texttt{of}\ \{\ldots\} \mapsto e & \gg_P & e \\
\texttt{case}\ e_r\ \texttt{of}\ \{\texttt{f} = p, \Delta, \ldots\} \mapsto e & \gg_P & (\lambda p.e)(\#\texttt{f}\ (e_r)), \\
 & & \text{se } \texttt{case}\ e_r\ \texttt{of}\ \{\Delta, \ldots\} \gg_P e \\
\{e|\} & \gg_M & e \\
\{e\,|\,{+}\texttt{f} \leftarrow e_f\ ,\ \Delta\} & \gg_M & \#+\texttt{f}\quad (e_f)\quad \{e\,|\,\Delta\} \\
\{e\,|\,{-}\texttt{f}\ ,\ \Delta\} & \gg_M & \#-\texttt{f}\quad \{e\,|\,\Delta\} \\
\{e\,|\,\texttt{f} \leftarrow e_f\ ,\ \Delta\} & \gg_M & \#+\texttt{f}\quad (e_f)\quad (\#-\texttt{f}\quad \{e\,|\,\Delta\})
\end{array}
$$

Portanto, para a definição do sistema de tipos, basta considerarmos as funções de manipulação de registro.

5.3 Sistema de tipos

Esta seção descreve as extensões propostas para que o Sistema CT suporte polimorfismo de registro.

Expressões de termos

A única mudança exigida na sintaxe de expressões de termos é a inclusão do registro vazio `{}` e das operações de registro primitivas:

Variáveis de termos (símbolos)	$x \in \mathbf{X}$	
Nomes de campos	$\mathrm{f} \in \mathbf{F}$	
Programas	$p \in \mathbf{P}$	$::= e \mid \texttt{leto}\ x = e\ \texttt{in}\ p$
Expressões	$e \in \mathbf{E}$	$::= x \mid \lambda x.\, e \mid\ e\, e' \mid\ \texttt{let}\ x = e\ \texttt{in}\ e' \mid$
		$\{\} \mid \#\texttt{f} \mid \#+\texttt{f} \mid \#-\texttt{f}$

O tipo registro

Ao invés de codificar registros como construtores de dados com múltiplos parâmetros, acrescentaremos um tipo primitivo de registro ao sistema de tipos. Os tipos simples são estendidos, como se segue:

$$\tau \;=\; \ldots \mid \{\mathrm{f}_1 : \tau_1, \ldots, \mathrm{f}_n : \tau_n\} \text{ para } n \geq 0$$

Nenhuma mudança é necessária ao sistema de *kinds*. O *kind* de um registro é $\star$. Como valores de registro podem ser construídos a partir de uma sequência de operações primitivas de extensão sobre o registro vazio, não precisamos de uma regra de tipagem para todos os registros, mas apenas para o registro vazio:

$$\frac{}{\Gamma \vdash \{\} : \{\}} \quad \text{(REC)}$$

É necessária uma mudança na definição da função lcg para que suporte o construtor de tipos registro. Esta mudança consiste no acréscimo de uma cláusula à definição da função auxiliar lcg′. Considerando-se que a notação $\{f_1, \ldots, f_n\}$ representa o conjunto de nomes de campos do tipo registro $\{\mathrm{f}_1 : \tau_1, \ldots, \mathrm{f}_n : \tau_n\}$, a figura 5.2 apresenta a nova definição da função lcg′, com uma nova para tratar registros.

Expressando o polimorfismo de registros: alternativas

O acréscimo de um tipo registro *leve* ao sistema de tipos é o primeiro passo para a construção de um sistema de tipos com suporte a polimorfismo de registro. O próximo é projetar o sistema de tipos de tal maneira que possa ser possível atribuir tipos às operações primitivas de manipulação de registros. Estes tipos devem ser tais que certas relações entre as variáveis de tipo dos mesmos possam ser expressas:

1. uma variável de tipo só pode ser instanciada para um tipo registro;
2. uma variável de tipo só pode ser instanciada para um tipo registro que possua os campos f_1, f_2 etc.;
3. uma variável de tipo só pode ser instanciada para um tipo registro que não tenha os campos f_1, f_2 etc.;
4. um par de variáveis só pode ser instanciado para tipos registro, que devem diferir apenas quanto à presença de um campo.

Existem muitas maneiras de expressar esses relacionamentos:

- **Subtipagem**: Subtipagem é a maneira mais antiga de se expressar polimorfismo de registro, sendo comumente usada em linguagens orientadas a objeto. Através de uma relação de ordem parcial entre os tipos, é possível expressar a informação de que uma variável só pode ser instanciada para um registro que contenha um certo conjunto de campos, ou seja, os relacionamentos 1 e 2 definidos acima. Porém, subtipagem não pode expressar informação negativa, como por exemplo nos relacionamentos 3 e 4 descritos acima. Usando subtipagem, o tipo da função de projeção para o campo `field` seria:

  $$\texttt{\#field} :: \quad \forall \texttt{a} \preceq \texttt{\{field::b\}.a} \rightarrow \texttt{b}$$

  Os tipos de outras operações primitivas de registro (restrição, extensão) não podem ser expressas usando subtipagem. No entanto, uma operação de atualização de campo que não muda o tipo do campo pode ser expressa.

$$
\begin{array}{rcl}
\mathrm{lcg}(\tau_1, \tau_2) & = & \tau \\
 & & \text{onde } (\tau, S_1, S_2, V) = \mathrm{lcg}'(\tau_1, \tau_2, \mathrm{id}, \mathrm{id}, \mathrm{tv}(\tau) \cup \mathrm{tv}(\tau')) \\
\mathrm{lcg}'(\mathrm{C}, \tau, S_1, S_2, V) & = & \begin{cases} (\mathrm{C}, S_1, S_2, V) & \text{se } \tau = \mathrm{C} \\ \mathrm{gen}(\mathrm{C}, \tau, S_1, S_2, V) & \text{caso contrário} \end{cases} \\
\mathrm{lcg}'(\tau_1\,\tau_2, \tau_1'\,\tau_2', \mathrm{S_1}, \mathrm{S_2}, \mathrm{V}) & = & \begin{cases} (\tau\,\tau', S, S', V_2) \\ \qquad \text{se } \mathrm{kind_T}(\tau_1) = \mathrm{kind_T}(\tau_1'), \\ \qquad\quad (\tau, S_1', S_2', V_1) = \mathrm{lcg}'(\tau_1, \tau_1', S_1, S_2, V), \\ \qquad\quad (\tau', S, S', V_2) = \mathrm{lcg}'(\tau_2, \tau_2', S_1', S_2', V_1) \\ \mathrm{gen}(\tau_1\,\tau_2, \tau_1'\,\tau_2', S_1, S_2, V), \text{ caso contrário} \end{cases} \\
\mathrm{lcg}'\left(\begin{array}{c} \tau = \{\mathrm{f}_1 : \tau_1, \ldots, \mathrm{f}_n : \tau_n\}, \\ \tau' = \{\mathrm{f}_1' : \tau_1', \ldots, \mathrm{f}_m' : \tau_m'\}, \\ S_1, S_2, V \end{array} \right) & = & \begin{cases} (\tau'', S_n^1, S_n^2, V_n) & \text{se } m = n \text{ e} \\ & \{\mathrm{f}_1, \ldots, \mathrm{f_n}\} = \{\mathrm{f}_1', \ldots, \mathrm{f}_m'\} \\ \mathrm{gen}(\tau, \tau', S_1, S_2, V) & \text{caso contrário} \end{cases} \\
\text{onde } \tau'' & = & \{\mathrm{f}_1 : \tau_1'', \ldots, \mathrm{f}_n : \tau_n''\}, \text{ e para } 1 \leq i \leq n : \\
(\tau_i'', S_i^1, S_i^2, V_i) & = & \mathrm{lcg}'(\tau_i, \mathrm{sel}(\mathrm{f}_i), S_{i-1}^1, S_{i-1}^2, V_{i-1}), \\
S_0^1 & = & S_1 \\
S_0^2 & = & S_2 \\
V_0 & = & V \\
\mathrm{sel}(\mathrm{f}) & = & \tau_j' \text{ se } \mathrm{f}_j' = \mathrm{f}, \text{ caso contrário indefinido} \\
\text{em outros casos de } \tau_1 \text{ e } \tau_2 : & & \\
\mathrm{lcg}'(\tau_1, \tau_2, S_1, S_2, V) & = & \mathrm{gen}(\tau_1, \tau_2, S_1, S_2, V) \\
\\
\mathrm{gen}(\tau_1, \tau_2, S_1, S_2, V) & = & \begin{cases} (\alpha, S_1, S_2, V) \\ \qquad \text{se } S_1(\alpha) = \tau_1, S_2(\alpha) = \tau_2, \text{ para algum } \alpha \\ (\alpha', S_1' \circ S_1, S_2 \circ S_2, V \cup \{\alpha'\}) \\ \qquad \text{caso contrário, onde } \alpha' \notin V, \\ \qquad\quad S_1' = [\alpha' \mapsto \tau_1],\ S_2' = [\alpha' \mapsto \tau_2] \end{cases}
\end{array}
$$

Figura 5.2: A nova função *lcg*.

- ***Kinds* de registro**: Em [32], Ohori acrescenta *kinds* de registro (*record kinds*) ao sistema de *kinds*. Estes *kinds* são usados para expressar a informação de que uma variável de tipo só pode ser instanciada para tipos registro e que certos campos, com nomes e tipos específicos, devem estar presentes no tipo instanciado. Assim como em subtipagem, esta abordagem não permite a expressão de informações negativas sobre a presença de campos em um registro, ou sobre a relação entre campos de diferentes registros. Usando *kinds* de registro, o tipo de uma operação de atualização de registro que não muda o tipo do campo seria:

$$\texttt{updateField} :: \quad \sigma \rightarrow \alpha^{\{\!\{\text{field}:\sigma\}\!\}} \rightarrow \alpha$$

- **Variáveis de linha**: Propostas mais recentes usam variáveis de linha (*row variables*) para expressar polimorfismo de registro. Variáveis de linha, apesar de presentes em uma expressão de tipo, não representam tipos, e sim linhas, que são conjuntos de campos de registro. Com variáveis de linha é possível atribuir um tipo a todas as operações primitivas de registro (projeção, extensão e restrição). Usando variáveis de linha, a operação de extensão de registros tem o seguinte tipo:

$$\texttt{\#+field} :: \quad \tau \rightarrow \texttt{\{ r \}} \rightarrow \texttt{\{ field :: } \tau \texttt{ | r \}}$$

- **Restrições de tipo**: A proposta deste documento faz uso de restrições de tipo (*type constraints*) para expressar polimorfismo de registro. Esta abordagem tem a vantagem de não exigir alterações no sistema de *kinds*, bastando o acréscimo de dois novos tipos de restrição. Esta abordagem será descrita na próxima seção.

Restrições de registro

Como visto anteriormente, para que o sistema de tipos suporte as três operações primitivas de registro (projeção, extensão e restrição), ele deve ser capaz de expressar os seguintes predicados ou restrições sobre variáveis de tipos:

1. A variável só pode ser instanciada para tipos registro;
2. Os tipos registro para os quais a variável pode ser instanciada devem incluir certos campos;
3. Os tipos registro para os quais a variável pode ser instanciada não podem incluir certos campos;
4. Os tipos registro para os quais duas variáveis podem ser instanciadas devem ter os mesmos campos, com exceção de um certo campo, que deve estar contido apenas em um dos registros.

Uma única restrição pode ser usada para expressar estas propriedades. A restrição *estende* é escrita como:

`+field` :: $\tau_{fld} \rightarrow \tau_{src} \rightarrow \tau_{tgt}$

Esta restrição só pode ser satisfeita se τ_{src} for um registro que não possui o campo `field` e τ_{tgt} for um registro que tem os mesmos campos que τ_{src} porém com o acréscimo de um campo chamado `field` com tipo τ_{fld}.

Quando uma variável de tipo aparece como τ_{src} ou τ_{tgt}, ela só pode ser instanciada para um tipo registro. Tais variáveis são chamadas *variáveis de registro*.

Usando a restrição *estende*, as operações básicas de registro têm os seguintes tipos, para um dado campo `f`:

- `#f` :: `{ +f` :: $\tau \rightarrow \alpha \rightarrow \beta$ `}` . $\beta \rightarrow \tau$

  A restrição faz com que β só possa ser instanciada para tipos registro que possuem um campo chamado `f` com tipo τ. O resultado da função é um valor de tipo τ (o tipo do campo `f`).

- `#+f` :: `{ +f` :: $\tau \rightarrow \alpha \rightarrow \beta$ `}` . $\tau \rightarrow \alpha \rightarrow \beta$

  A restrição faz com que as variáveis α e β só possam ser instanciadas para tipos registro τ_α e τ_β , que diferem apenas quanto à presença do campo `f`, com tipo τ, que deve estar presente em τ_β, porém não em τ_α.

- `#-f` :: `{ +f` :: $\tau \rightarrow \alpha \rightarrow \beta$ `}` . $\beta \rightarrow \alpha$

  A restrição faz com que as variáveis α e β só possam ser instanciadas para tipos registro que diferem apenas quanto à presença do campo `f`, e que β deve ter o campo enquanto α deve omití-lo.

Restrições de registro negativas

Como visto acima, a restrição `+f` :: $\tau \rightarrow \alpha \rightarrow \beta$ codifica os relacionamentos:

- α só pode ser instanciada para tipos registro que não possuem o campo de nome `f`;
- β só pode ser instanciada para tipos registro que possuem o campo de nome `f` e tipo τ;
- os tipos registro para os quais α e β forem instanciados só podem diferir quanto à presença do campo `f`.

Se o compilador permitir que o programador anote definições com restrições do tipo `+f` :: ..., torna-se fácil anotar uma definição com um tipo mais restritivo que o tipo inferido e

que restringe um parâmetro de forma que ele só possa ser registros que possuem um certo conjunto de campos. Para fazê-lo, basta o programador acrescentar as restrições

$$\begin{aligned} +\mathtt{f}_1 &:: \tau_1 \to \alpha_1 \to \beta \\ +\mathtt{f}_2 &:: \tau_2 \to \alpha_2 \to \alpha_1 \\ &\vdots \\ +\mathtt{f}_n &:: \tau_n \to \alpha_n \to \alpha_{n-1} \end{aligned}$$

onde $\mathtt{f}_i :: \tau_i$ são os campos que o parâmetro correspondente à variável β deve ter, e $\{\alpha_i\}$ é um conjunto de variáveis novas.

Por exemplo, o tipo inferido para a seguinte função é $\forall\alpha.\alpha \to (\alpha, \alpha)$:

```
func x = (x,x)
```

Para restringi-la de forma que só possa ser usada com registros que possuam um campo com nome `f` com tipo qualquer, podemos anotá-la com o seguinte tipo:

$$\forall\alpha.\beta_1.\beta_2.\{+\mathtt{f} :: \beta_1 \to \beta_2 \to \alpha\}.\alpha \to (\alpha, \alpha)$$

Como, porém, pode o programador na mesma situação acrescentar uma restrição *negativa* à variável α? A resposta óbvia é usar uma estratégia similar, acrescentando a restrição $+\mathtt{f} :: \beta_1 \to \alpha \to \beta_2$, com variáveis novas β_1 e β_2.

A anotação do tipo de `func` seria:

$$\forall\alpha\beta_1\beta_2.\{+\mathtt{f} :: \beta_1 \to \alpha \to \beta_2\}.\alpha \to (\alpha, \alpha)$$

Note o que acontece quando instanciamos a variável α nas anotações de tipo mostradas acima: primeiramente, suponha que instanciamos α para $\{\mathtt{f} : \mathrm{Int}\}$ na primeira anotação de tipo:

$$\begin{aligned} &\forall\beta_1\beta_2.[\alpha \mapsto \{\mathtt{f} : \mathrm{Int}\}]\{+\mathtt{f} :: \beta_1 \to \beta_2 \to \alpha\}.\alpha \to (\alpha, \alpha) = \\ &\forall\beta_1\beta_2.\{+\mathtt{f} :: \beta_1 \to \beta_2 \to \{\mathtt{f} : \mathrm{Int}\}\}.\{\mathtt{f} : \mathrm{Int}\} \to (\{\mathtt{f} : \mathrm{Int}\}, \{\mathtt{f} : \mathrm{Int}\}) \end{aligned}$$

como as instâncias de β_2 só podem diferir de $\{\mathrm{f} : \mathrm{Int}\}$ pela ausência do campo `f`, a única possível substituição para β_2 é $[\beta_2 \mapsto \{\}]$. De modo semelhante, a única possível substituição para β_1 é $[\beta_1 \mapsto \mathrm{Int}]$. Fazendo as duas substituições, chegamos ao tipo:

$$\{+\mathtt{f} :: \mathrm{Int} \to \{\} \to \{\mathtt{f} : \mathrm{Int}\}\}.\{\mathtt{f} : \mathrm{Int}\} \to (\{\mathtt{f} : \mathrm{Int}\}, \{\mathtt{f} : \mathrm{Int}\})$$

Como a restrição está resolvida, podemos omití-la, resultando no tipo:

$$\{\mathtt{f} : \mathrm{Int}\} \to (\{\mathtt{f} : \mathrm{Int}\}, \{\mathtt{f} : \mathrm{Int}\})$$

.

Se tentarmos fazer a mesma instanciação na segunda anotação de tipo, substituindo α por $\{g : \text{Int}\}$, por exemplo, teremos o seguinte tipo como resultado:

$$\forall\beta_1\beta_2.\{+\texttt{f} :: \beta_1 \to \{\texttt{g} : \text{Int}\} \to \beta_2\}.\{\texttt{g} : \text{Int}\} \to (\{\texttt{g} : \text{Int}\}, \{\texttt{g} : \text{Int}\})$$

Como no caso anterior, podemos usar a relação entre o tipo registro $\{g : \text{Int}\}$ e a variável β_2 para especializar mais o tipo, que se torna:

$$\forall\beta_1.\{+\texttt{f} :: \beta_1 \to \{\texttt{g} : \text{Int}\} \to \{\texttt{g} : \text{Int}, \texttt{f} : \beta_1\}\}.\{\texttt{g} : \text{Int}\} \to (\{\texttt{g} : \text{Int}\}, \{\texttt{g} : \text{Int}\})$$

Diferentemente do caso anterior, não é possível deduzir uma substituição para β_1, pois qualquer substituição seria válida.

Como todas as variáveis de tipo que aparecem do lado direito do tipo já foram instanciadas, não há como decidir qual tipo a variável β_1 pode assumir. Portanto, o tipo resultante é ambíguo.

Para remover esta limitação, e permitir que o programador possa expressar informações negativas arbitrariamente, introduzimos a restrição negativa

$$-\texttt{f} :: \alpha$$

que significa que α só pode ser instanciada para um tipo registro que não possui o campo `f`.

Podemos entender esta restrição como uma versão *existencial* da restrição `+f` :

$$-\texttt{f} :: \alpha \equiv \exists\beta_1\beta_2.+\texttt{f} :: \beta_1 \to \alpha \to \beta_2$$

Usando esta restrição, o segundo tipo anotado se torna

$$\forall\alpha.\{-\texttt{f} :: \alpha\}.\alpha \to (\alpha, \alpha)$$

evitando assim a menção às variáveis β_1 e β_2. Substituindo α por $\{\texttt{g} : \text{Int}\}$ resulta em:

$$\{-\texttt{f} :: \{g : \text{Int}\}\}.\{\texttt{g} : \text{Int}\} \to (\{\texttt{g} : \text{Int}\}, \{\texttt{g} : \text{Int}\})$$

que pode ser simplificado para:

$$\{\texttt{g} : \text{Int}\} \to (\{\texttt{g} : \text{Int}\}, \{\texttt{g} : \text{Int}\})$$

que não é ambíguo.

Estes dois tipos de restrição ($+\texttt{f}$ e $-\texttt{f}$) são a r proposta para a sintaxe de restrições do Sistema CT. A primeira restrição é chamada restrição positiva, ou restrição *estende*, enquanto que a segunda é chamada restrição negativa, ou restrição *omite*.

A sintaxe de tipos do Sistema CT com restrições de registro

A sintaxe de tipos do Sistema CT, com o acréscimo dos dois tipos de restrição novos, é mostrada abaixo:

Kinds	$K \in \mathbf{Kind}$	$::= \star \mid K \to K$	
Variáveis de tipo	$\alpha, \beta, \gamma \in \mathbf{V}$		
Conjuntos de variáveis	$V, \bar{\alpha}, \bar{\beta}, \bar{\gamma} \subseteq \mathbf{V}$		
Construtores de tipos	$C \in \mathbf{C}$		
Expressões de tipo simples	$\tau \in \mathbf{T}$	$::= \alpha \mid C \mid \tau_1\, \tau_2 \mid$	
		$\{\mathrm{f}_1 : \tau_1, \ldots, \mathrm{f}_n : \tau_n\}$	$n \geq 0$
Conjuntos de restrições	$\kappa \in \mathbf{K}$	$::= \{c_1, \ldots, c_n\}$	$n \geq 0$
Restrições	c	$::= x : \tau \mid$ `+f` $: \tau \mid$ `-f` $: \tau$	
Tipos	$\sigma \in \mathbf{J}$	$::= \forall \bar{\alpha}.\kappa.\tau$	

Regras de tipagem para o Sistema CT com restrições de registro

A figura 5.3 mostra quatro novas regras de tipagem (PROJ, EXT, REST e REC) que são acrescentadas ao Sistema CT para dar suporte a polimorfismo de registro.

Satisfazibilidade de restrições de registro

Assim como a sintaxe de conjuntos de restrições foi alterada para incluir as restrições de registro, novas regras de satisfazibilidade precisam ser adicionadas para tratar as novas restrições. As novas regras, $\mathrm{SAT}_{estende}$ e SAT_{omite}, são apresentadas juntamente com as regras originais na figura 5.4.

A regra SAT_{omite} simplesmente diz que uma restrição *omite* $-\mathrm{f} : \tau$ só pode ser satisfeita se τ for um tipo registro que não contém uma campo de nome f, enquanto que a regra $\mathrm{SAT}_{estende}$ requer que, para que uma restrição $+\mathrm{f} : \tau$ seja satisfeita, τ precisa ter a forma $\tau_f \to \tau_s \to \tau_t$, onde τ_s e τ_τ são registros com o mesmo conjunto de campos, com a exceção do campo de nome f e tipo τ_f, que deve estar presente em τ_t porém não em τ_s.

5.4 Inferência de tipos

Restrições de registro e substituições principais

Como vimos no capítulo anterior, o processo de inferência de tipos do Sistema CT requer o cálculo de uma *substituição principal* para cada conjunto de restrições que aparece no contexto de tipagem. Estas substituições, quando feitas nos tipos restringidos correspondentes, resulta na generalização mínima dos tipos que podem ser derivados dos mesmos, respectivamente.

Uma propriedade importante de substituições principais, que pode ser derivada das regras SAT_{muitas} e $\mathrm{SAT}_{\emptyset}$, é que a substituição principal de uma união de dois conjuntos de

$$\frac{|\Gamma(x)| \geq 2 \quad \tau = \bigvee\{\tau' \mid \forall\bar{\alpha}.\kappa.\tau' \in \Gamma(x)\} \quad \bar{\alpha} = \mathrm{tv}(\tau)}{\Gamma \vdash x : \forall\bar{\alpha}.\{x : \tau\}.\tau} \quad (\mathrm{VAR}_{\mathrm{many}})$$

$$\frac{\Gamma(x) = \{\sigma\}}{\Gamma \vdash x \ : \ \sigma} \quad (\mathrm{VAR}_1)$$

$$\frac{\Gamma \vdash x : \forall\bar{\alpha}.\kappa.\tau \quad \mathrm{dom}(S) \subseteq \bar{\alpha} \quad \Gamma_S \models \kappa}{\Gamma \vdash x : S\kappa.\, S\tau} \quad (\mathrm{INST})$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \kappa_1.\tau_1 \quad \sigma_1 = \forall\bar{\alpha}.\kappa_1.\tau_1 \quad \bar{\alpha} = \mathrm{tv}(\kappa_1.\tau_1) - \mathrm{tv}(\Gamma) \\ \Gamma^x \cup \{x : \sigma_1\} \vdash e_2 : \kappa_2.\tau_2\end{array}}{\Gamma \vdash \mathrm{let}\ x = e_1\ \mathrm{in}\ e_2 : \kappa_2.\tau_2} \quad (\mathrm{LET})$$

$$\frac{\Gamma, \{u : \tau'\} \vdash e : \kappa.\tau}{\Gamma \vdash \lambda u.e : \kappa.\tau' \to \tau} \quad (\mathrm{ABS})$$

$$\frac{\Gamma \vdash e_1 : \kappa_1.\tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \kappa_2.\tau_2 \quad \Gamma_S \models (\kappa_1 \cup \kappa_2)}{\Gamma \vdash e_1 e_2 : S(\kappa_1 \cup \kappa_2.\tau_2)} \quad (\mathrm{APPL})$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \kappa_1.\tau_1 \quad \sigma_1 = \forall\bar{\alpha}.\kappa_1.\tau_1 \quad \bar{\alpha} = \mathrm{tv}(\kappa_1.\tau_1) - \mathrm{tv}(\Gamma) \\ \Gamma \cup \{x : \sigma_1\} \vdash p : \kappa_2.\tau_2\end{array}}{\Gamma \vdash \mathrm{leto}\, x = e_1\, \mathrm{in}\, p \ : \ \kappa_2.\tau_2} \quad (\mathrm{LETO})$$

$$\frac{}{\Gamma \vdash \{\} \ : \ \{\}} \quad (\mathrm{REC})$$

$$\frac{}{\Gamma \vdash \#\mathrm{f} \ : \ \{+\mathrm{f} \ : \ \alpha_1 \to \alpha_2 \to \alpha_3\}.\, \alpha_3 \to \alpha_1} \quad (\mathrm{PROJ})$$

$$\frac{}{\Gamma \vdash \# + \mathrm{f} \ : \ \{+\mathrm{f} \ : \ \alpha_1 \to \alpha_2 \to \alpha_3\}.\, \alpha_1 \to \alpha_2 \to \alpha_3} \quad (\mathrm{EXT})$$

$$\frac{}{\Gamma \vdash \# - \mathrm{f} \ : \ \{+\mathrm{f} \ : \ \alpha_1 \to \alpha_2 \to \alpha_3\}.\, \alpha_3 \to \alpha_2} \quad (\mathrm{REST})$$

Figura 5.3: Sistema de tipos do Sistema CT com polimorfismo de registro.

$$\frac{\begin{array}{c} S\tau_s = \{f_1 : \tau_1, \ldots, f_n : \tau_n\} \text{ para } n \geq 0 \\ S\tau_t = \{f_1 : \tau_1, \ldots, f_n : \tau_n, f : S\tau_f\} \\ f \neq f_i, \text{ para } 1 \leq i \leq n \end{array}}{\Gamma \models_S +\mathtt{f} : \tau_f \to \tau_s \to \tau_t} \quad (\text{SAT}_{estende})$$

$$\frac{\begin{array}{c} S\tau = \{f_1 : \tau_1, \ldots, f_n : \tau_n\} \text{ para } n \geq 0 \\ f \neq f_i \text{ para } 1 \leq i \leq n \end{array}}{\Gamma \models_S -\mathtt{f} : \tau} \quad (\text{SAT}_{omite})$$

$$\frac{S\tau = S\tau' \quad \text{dom}(S) \subseteq (\bar{\alpha} \cup \text{tv}(\tau)) - \text{tv}(\Gamma) \quad \Gamma \models_{S'} S\kappa}{\Gamma \cup \{x : \forall \bar{\alpha}.\kappa.\tau'\} \models_{S' \circ S} x : \tau} \quad (\text{SAT}_1)$$

$$\frac{}{\Gamma \models_S \emptyset} \quad (\text{SAT}_\emptyset)$$

$$\frac{\Gamma \models_S x : \tau \qquad \Gamma \models_S \kappa}{\Gamma \models_S \{x : \tau\} \cup \kappa} \quad (\text{SAT}_{muitas})$$

Figura 5.4: Regras de satisfazibilidade com restrições de registro.

restrições pode ser calculada com os seguintes passos: computa-se, primeiro, a substituição principal do primeiro conjunto; faz-se esta substituição no segundo conjunto e calcula-se a substituição principal do segundo conjunto, que, composta com a primeira substituição, formará a substituição principal da união dos conjuntos.

Formalmente, se $S^{\kappa,\Gamma}$ representa a substituição principal do conjunto de restrições κ em Γ, então:

$$S^{\kappa \cup \kappa',\Gamma} = S^{S^{\kappa,\Gamma}(\kappa'),\Gamma} \cdot S^{\kappa,\Gamma}$$

Uma consequência desta propriedade é que podemos usar dois algoritmos distintos para calcular a substituição principal dos conjuntos de restrições de registro e a substituição principal do conjunto de restrições normais.

Uma definição mais formal do problema do cálculo da substituição principal para um conjunto de restrições de registro será feita no próximo capítulo, onde também será apresentado um algoritmo para computá-la.

Capítulo 6

Conjuntos de campos

6.1 Definições e lemas

Como vimos no capítulo 5, um tipo registro é um conjunto de campos em que cada campo representa uma associação entre um *nome de campo* e um *tipo de campo*, onde cada nome é associado a apenas um campo. Esta definição é formalizada através do conceito de *conjuntos de campos*:

Definição 6.1. Um *conjunto de campos* é um conjunto finito $R \subseteq \mathbf{F} \times \mathbf{T}$, onde $\mathbf{F}$ é o conjunto de nomes de campos e $\mathbf{T}$ é o conjunto de tipos simples, que satisfaz a seguinte condição:

$$\text{se } \{(f, \tau), (f, \tau')\} \subseteq R \text{ então } \tau = \tau'$$

Representamos o conjunto de conjuntos de campos por $\mathbf{R}$ e usamos a meta-variável R, possivelmente indexada, para representar conjuntos de campos.

O relacionamento entre tipos registro e conjuntos de campos é definido pela *função de representação* e por sua inversa:

Definição 6.2. A função de representação $[\![\ ]\!] : \mathbf{T} \to \mathbf{R}$ é definida como:

$$\begin{aligned} [\![\{f_1 : \tau_1, \ldots, f_n : \tau_n\}]\!] &= \{(f_1, \tau_1), \ldots, (f_n, \tau_n)\} \\ &\quad \text{onde} \quad 1 \leq i < j \leq n \text{ implica que } f_i \neq f_j \end{aligned}$$

A sua inversa, $[\![\ ]\!]^{-1} : \mathbf{R} \to \mathbf{T}$, é uma função injetiva definida como:

$$\begin{aligned} [\![\{(f_1, \tau_1), \ldots, (f_n, \tau_n)\}]\!]^{-1} &= \{f_1 : \tau_1, \ldots, f_n : \tau_n\} \\ &\quad \text{onde} \quad 1 \leq i < j \leq n \text{ implica que } f_i \neq f_j \end{aligned}$$

Definição 6.3. Cada conjunto de campo tem um conjunto de nomes de campo correspondente, que é dado pela função fnames : $\mathbf{R} \to \mathcal{P}(\mathbf{F})$, definida como:

$$\text{fnames}(R) = \{f \mid (f, \tau) \in R \text{ para algum } \tau\}$$

Por economia de espaço, usaremos a abreviação $\text{fn}(R)$ para representar a expressão $\text{fnames}(R)$ quando necessário.

Definição 6.4. Dois conjuntos de campos R_1 e R_2 podem ter campos com o mesmo nome. A função cortypes : $\mathbf{R} \times \mathbf{R} \to \mathcal{P}(\mathbf{T} \times \mathbf{T})$, quando aplicada a R_1e R_2 nos dá o conjunto de pares de tipos que correspondem, respectivamente, aos tipos de campos com o mesmo nome em R_1 e R_2. Formalmente:

$$\text{cortypes}(R_1, R_2) = \{(\tau_1, \tau_2) \mid (f, \tau_1) \in R_1 \text{ e } (f, \tau_2) \in R_2, \text{ para algum } f\}$$

Dizemos que dois conjuntos de campos têm seus campos correspondentes equivalentes se os tipos dos campos correspondentes são iguais. Formalmente, se tomarmos $\Delta\mathbf{T} = \{(\tau, \tau) \mid \tau \in \mathbf{T}\}$, temos:

$$\begin{aligned} \text{cortypes}(R_1, R_2) &\subseteq \Delta\mathbf{T} \\ \text{onde } \Delta\mathbf{T} &= \{(\tau, \tau) \mid \tau \in \mathbf{T}\} \end{aligned}$$

Estas funções possuem algumas propriedades interessantes, que são mostradas nos seguintes lemas:

Lema 6.5. $\text{fnames}(\emptyset) = \emptyset$.

Demonstração.

$$\begin{aligned} \text{fnames}(\emptyset) &= \{f \mid (f, \tau) \in \emptyset, \text{ para algum } \tau\} \\ &= \emptyset \end{aligned}$$

□

Lema 6.6. *$R_1 \subseteq R_2$ se e somente se* $\text{fnames}(R_1) \subseteq \text{fnames}(R_2)$ *e* $\text{cortypes}(R_1, R_2) \subseteq \mathbf{\Delta T}$.

Demonstração. Para o sentido esquerda-direita, supomos que $R_1 \subseteq R_2$ e, por hipótese, que $\text{fnames}(R_1) \not\subseteq \text{fnames}(R_2)$. Neste caso teríamos um campo com nome f tal que $f \in \text{fnames}(R_1)$ porém $f \notin \text{fnames}(R_2)$.

Mas, se isto é verdade, então deve haver um campo (f, τ) que aparece em R_1 porém não em R_2, e portanto $R_1 \not\subseteq R_2$, o que contradiria nossa suposição.

Por outro lado, se $R_1 \subseteq R_2$ mas $\text{cortypes}(R_1, R_2) \not\subseteq \Delta\mathbf{T}$, então deve haver campos $(f, \tau_1) \in R_1$ e $(f, \tau_2) \in R_2$ tais que $\tau_1 \neq \tau_2$. Como estamos supondo que $R_1 \subseteq R_2$, e como $(f, \tau_1) \in R_1$, então deve ser o caso que $(f, \tau_1) \in R_2$. Pela condição encontrada na definição 6.1, teríamos que $\tau_1 = \tau_2$, o que é uma contradição.

Para a implicação inversa, supomos que $\text{fnames}(R_1) \subseteq \text{fnames}(R_2)$ e que $\text{cortypes}(R_1, R_2) \subseteq \Delta\mathbf{T}$. Se $R_1 \not\subseteq R_2$, então deve haver um campo $(f, \tau) \in R_1$ tal que $(f, \tau) \notin R_2$. Neste caso, uma de duas condições deve ser verdade: ou não há campos com nome f em R_2, ou existe um campo com nome f em R_2 cujo tipo não é τ. No primeiro caso, contradiríamos a suposição de que $\text{fnames}(R_1) \subseteq \text{fnames}(R_2)$, enquanto que no segundo caso, contradiríamos a suposição de que $\text{cortypes}(R_1, R_2) \subseteq \Delta\mathbf{T}$. □

Lema 6.7. $\text{fnames}(R_1 \cup R_2) = \text{fnames}(R_1) \cup \text{fnames}(R_2)$.

Demonstração.

$$\begin{aligned}
\text{fnames}(R_1 \cup R_2) &= \{f \mid (f,\tau) \in R_1 \cup R_2, \text{ para algum } \tau\} \\
&= \{f \mid (f,\tau) \in R_1, \text{ para algum } \tau\} \cup \{f \mid (f,\tau) \in R_2, \text{ para algum } \tau\} \\
&= \text{fnames}(R_1) \cup \text{fnames}(R_2)
\end{aligned}$$

□

Lema 6.8. $\text{fnames}(R_1 \cap R_2) \subseteq \text{fnames}(R_1) \cap \text{fnames}(R_2)$.

Demonstração.

$$\begin{aligned}
\text{fnames}(R_1 \cap R_2) &= \{f \mid (f,\tau) \in R_1 \cap R_2, \text{ para algum } \tau\} \\
&= \left\{ f \mid \begin{array}{l} (f,\tau_1) \in R_1, \text{ para algum } \tau_1, \text{ and} \\ \quad (f,\tau_2) \in R_3, \text{ para algum } \tau_2, \text{ and } \tau_1 = \tau_2 \end{array} \right\} \\
&\subseteq \{f \mid (f,\tau_1) \in R_1, \text{ para algum } \tau_1, \text{ and } (f,\tau_2) \in R_3, \text{ para algum } \tau_2\} \\
&\subseteq \{f \mid (f,\tau_1) \in R_1, \text{ para algum } \tau_1\} \cap \{f \mid (f,\tau_2) \in R_2, \text{ para algum } \tau_2\} \\
&\subseteq \text{fnames}(R_1) \cap \text{fnames}(R_2)
\end{aligned}$$

□

Lema 6.9. fnames *é monotônica com respeito a* $\subseteq$.

Demonstração. Para quaisquer dois conjuntos X e Y, se $X \subseteq Y$, então $Y = X \cup Y$ e vice-versa. No caso de conjuntos de campos, $R_1 \subseteq R_2$ se e somente se $R_1 \cup R_2 = R_2$.

Pelo lema 6.7, temos que

$$\begin{aligned}
\text{fnames}(R_2) &= \text{fnames}(R_1 \cup R_2) \\
&= \text{fnames}(R_1) \cup \text{fnames}(R_2)
\end{aligned}$$

o que significa que $\text{fnames}(R_1) \subseteq \text{fnames}(\text{R}_2)$. □

Lema 6.10. *Se* $R_1 \subseteq R_2$ *e* $(f,\tau) \in R_2$*, então* $(f,\tau) \notin R_1$ *se e somente se* $f \notin \text{fnames}(R_1)$.

Demonstração. Suponha, por hipótese, que $R_1 \subseteq R_2$, que $(f,\tau) \in R_2$ e que $(f,\tau) \notin R_1$. Suponha também que $f \in \text{fnames}(R_1)$.

Neste caso, deve existir um tipo τ' tal que $(f,\tau') \in R_1$. Como $R_1 \subseteq R_2$, então $(f,\tau') \in R_2$. Pela definição 6.1, se $(f,\tau) \in R_2$ e $(f,\tau') \in R_2$ então $\tau = \tau'$, o que significa que $(f,\tau) = (f,\tau') \in R_1$, o que por sua vez é uma contradição.

Por outro lado, suponha que $R_1 \subseteq R_2$, que $(f,\tau) \in R_2$ e que $f \notin \text{fnames}(R_1)$, mas que $(f,\tau) \in R_1$. Pela definição 6.3, como $(f,\tau) \in R_1$, então $f \in \text{fnames}(R_1)$, o que é uma contradição. □

Lema 6.11. *Se* $\text{fnames}(R_1) \cap \text{fnames}(R_2) = \emptyset$ *então* $R_1 \cap R_2 = \emptyset$.

Demonstração. Suponha que $\text{fnames}(R_1) \cap \text{fnames}(R_2) = \emptyset$ e que $R_1 \cap R_2 \neq \emptyset$. Teriam de haver, neste caso, f e τ tais que $(f, \tau) \in R_1$ e $(f, \tau) \in R_2$. Então, temos que $f \in \text{fnames}(R_1)$ e $f \in \text{fnames}(R_2)$ e, portanto, $\text{fnames}(R_1) \cap \text{fnames}(R_2) \neq \emptyset$, o que contradiria nossa suposição. □

Lema 6.12. *Se* $R_1 \subseteq R_3$, $R_2 \subseteq R_3$ *e* $R_1 \cap R_2 = \emptyset$, *então* $\text{fnames}(R_1) \cap \text{fnames}(R_2) = \emptyset$.

Demonstração. Suponha que $R_1 \subseteq R_3$, $R_2 \subseteq R_3$ e $R_1 \cap R_2 = \emptyset$, mas que $\text{fnames}(R_1) \cap \text{fnames}(R_2) \neq \emptyset$. Devem haver f, τ e τ' tais que $(f, \tau) \in R_1$ e $(f, \tau') \in R_2$. Isto implica que $\{(f, \tau), (f, \tau')\} \subseteq R_3$, o que significa, pela definição 6.1, que $\tau = \tau'$. Como $(f, \tau) \in R_1$ e $(f, \tau) = (f, \tau') \in R_2$, então $(f, \tau) \in R_1 \cap R_2$, o que implica que $R_1 \cap R_2 \neq \emptyset$, contradizendo nossa hipótese. □

Lema 6.13. $\text{fnames}(R_1) - \text{fnames}(R_2) \subseteq \text{fnames}(R_1 - R_2)$.

Demonstração. Se supormos que $f \in \text{fnames}(R_1) - \text{fnames}(R_2)$, então $f \in \text{fnames}(R_1)$ mas $f \notin \text{fnames}(R_2)$. Pela definição 6.3, deve haver um τ tal que $(f, \tau) \in R_1$ e não pode haver um τ' tal que $(f, \tau') \in R_2$. Como $(f, \tau) \in R_1$ mas $(f, \tau) \notin R_2$, então $(f, \tau) \in R_1 - R_2$ e $f \in \text{fnames}(R_1 - R_2)$. □

Lema 6.14. *Se* $\text{cortypes}(R_1, R_2) \subseteq \Delta\mathbf{T}$ *então* $\text{fnames}(R_1 - R_2) = \text{fnames}(R_1) - \text{fnames}(R_2)$.

Demonstração. Pelo lema 6.13, $\text{fnames}(R_1 - R_2) \supseteq \text{fnames}(R_1) - \text{fnames}(R_2)$. Suponhamos que $\text{cortypes}(R_1, R_2) \subseteq \Delta\mathbf{T}$, e que $\text{fnames}(R_1 - R_2) \neq \text{fnames}(R_1) - \text{fnames}(R_2)$. Deve ser o caso, então, que $\text{fnames}(R_1 - R_2) \supset \text{fnames}(R_1) - \text{fnames}(R_2)$.

Isto implica que deve haver um f tal que $f \in \text{fnames}(R_1 - R_2)$ e que $f \notin \text{fnames}(R_1) - \text{fnames}(R_2)$. A primeira proposição implica que deve haver um τ tal que $(f, \tau) \in R_1 - R_2$, o que significa que $(f, \tau) \in R_1$ e que $(f, \tau) \notin R_2$. Como $(f, \tau) \in R_1$, a segunda proposição implica que deve haver um τ' tal que $(f, \tau') \in R_2$. Mas, como $(f, \tau) \notin R_2$, então $\tau \neq \tau'$.

Teríamos então que $(f, \tau) \in R_1$ e que $(f, \tau') \in R_2$, implicando que $(\tau, \tau') \in \text{cortypes}(R_1, R_2)$. Mas, como $\text{cortypes}(R_1, R_2) \subseteq \Delta\mathbf{T}$, então $\tau = \tau'$, o que seria uma contradição. □

Lema 6.15. *Se* R_1 *e* R_2 *são conjuntos de campos, então* $R_1 \cup R_2$ *é um conjunto de campos se e somente se* $\text{cortypes}(R_1, R_2) \subseteq \Delta\mathbf{T}$.

Demonstração. Suponha que R_1, R_2e $R_1 \cup R_2$ são conjuntos de campos, mas que $\text{cortypes}(R_1, R_2) \not\subseteq \Delta\mathbf{T}$. Devem haver, então, f, τ e τ' tais que $(f, \tau) \in R_1$, $(f, \tau') \in R_2$, e $\tau \neq \tau'$. Neste caso, $\{(f, \tau), (f, \tau')\} \subseteq R_1 \cup R_2$. Como $R_1 \cup R_2$ deve ser um conjunto de campos, então, pela definição 6.1, temos que $\tau = \tau'$, o que contradiz a hipótese.

Por outro lado, se supormos que R_1 e R_2 são conjuntos de campos e que $\text{cortypes}(R_1, R_2) \subseteq \Delta\mathbf{T}$, mas que $R_1 \cup R_2$ não é um conjunto de campos, então devem haver f, τ e τ' tais que $\{(f, \tau), (f, \tau')\} \subseteq R_1 \cup R_2$, e que $\tau \neq \tau'$. Como R_1e R_2 são conjuntos de campos, não pode ser o caso que $\{(f, \tau), (f, \tau')\} \subseteq R_1$ ou que $\{(f, \tau), (f, \tau')\} \subseteq R_2$.

Suponhamos então, sem perda de generalidade, que $(f, \tau) \in R_1$ e que $(f, \tau') \in R_2$.

Pela definição 6.4, $(\tau, \tau') \in \text{cortypes}(R_1, R_2)$. Entretanto, como $\text{cortypes}(R_1, R_2) \subseteq \Delta\mathbf{T}$, então $\tau = \tau'$, o que é uma contradição. □

Lema 6.16. *Se R_1 e R_2 são conjuntos de campos, então $R_1 - R_2$ é um conjunto de campos.*

Demonstração. Se $\{(f,\tau),(f,\tau')\} \subseteq R_1 - R_2$, então $\{(f,\tau),(f,\tau')\} \subseteq R_1$. Pela definição 6.1, teríamos que $\tau = \tau'$. Novamente, pela definição 6.1, teríamos que $R_1 - R_2$ é um conjunto de campos. □

6.2 Operações sobre conjuntos de campos

Três operações parciais sobre conjuntos de campos serão usadas para definir o problema do cálculo da substituição principal de um conjunto de restrições de registro: a operação de concatenação de conjuntos de campos ($\oplus$); a operação de diferença de conjuntos de campos ($\ominus$); e a operação de remoção de campos, definidas a seguir:

Definição 6.17. O operador de conjuntos de campos $\oplus : \mathbf{R} \times \mathbf{R} \to \mathbf{R}$ é um operador parcial definido para pares de conjuntos de campos cujos conjuntos de nomes correspondentes são disjuntos. O seu resultado é simplesmente a união dos dois conjuntos. Formalmente:

$$R_1 \oplus R_2 \quad = \quad R_1 \cup R_2 \text{ se } \mathrm{fnames}(R_1) \cap \mathrm{fnames}(R_2) = \emptyset$$

Definição 6.18. O operador de diferença de conjuntos de campos $\ominus : \mathbf{R} \times \mathbf{R} \to \mathbf{R}$ é um operador parcial definido para pares de conjuntos de campos onde o segundo conjunto de campo é um subconjunto do primeiro. O resultado da operação é a diferença comum de conjuntos dos dois conjuntos. Formalmente:

$$R_1 \ominus R_2 = R_1 - R_2 \text{ se } R_2 \subseteq R_1$$

Definição 6.19. A operação de remoção de campos $(/) : \mathbf{R} \times \mathbf{F} \to \mathbf{R}$ é definida como:

$$R/F = \{(f,\tau) \mid (f,\tau) \in R \text{ e } f \notin F\}$$

Algumas das propriedades destas operações são demonstradas nos seguintes lemas:

Lema 6.20. *Se $R_1 \ominus R_2$ é definido, então $R_1 \ominus R_2 = R_1/\mathrm{fnames}(R_2)$.*

Demonstração. Se $R_1 \ominus R_2$ é definido, então temos, pela definição 6.18, que $R_2 \subseteq R_1$ e que $R_1 \ominus R_2 = R_1 - R_2$. Portanto, temos:

$$R_1 - R_2 \quad = \quad \{(f,\tau) \mid (f,\tau) \in R_1 \text{ e } (f,\tau) \notin R_2\}$$

Mas, como $R_2 \subseteq R_1$, temos pelo lema 6.10 que a proposição $(f,\tau) \in R_1$ e $(f,\tau) \notin R_2$ são intercambiáveis com as proposições $(f,\tau) \in R_1$ e $f \notin \mathrm{fnames}(R_2)$. Fazendo esta substituição e usando a definição 6.19, a expressão acima se torna:

$$\begin{aligned} R_1 - R_2 &= \{(f,\tau) \mid (f,\tau) \in R_1 \text{ e } f \notin \mathrm{fnames}(R_2)\} \\ &= R_1/\mathrm{fnames}(R_2) \end{aligned}$$

□

Lema 6.21. $\text{fnames}(R/F) = \text{fnames}(R) - F$.

Demonstração. Pelas definições 6.3 e 6.19, temos:

$$\begin{aligned}
\text{fnames}(R/F) &= \{f \mid (f,\tau) \in R/F, \text{ para algum } \tau\} \\
&= \{f \mid (f,\tau) \in \{(f',\tau') \mid (f',\tau') \in R \text{ e } f' \notin F\}, \text{ para algum } \tau\} \\
&= \{f' \mid (f',\tau') \in R \text{ e } f' \notin F\} \\
&= \{f' \mid (f',\tau') \in R\} - F \\
&= \text{fnames}(R) - F
\end{aligned}$$

□

Lema 6.22. *O conjunto vazio é uma identidade de* $\oplus$.

Demonstração. Pelo lema 6.5, temos:

$$\text{fnames}(R) \cap \text{fnames}(\emptyset) = \text{fnames}(R) \cap \emptyset = \emptyset$$

Pela definição 6.17, temos:

$$R \oplus \emptyset = R \cup \emptyset = R = \emptyset \cup R = \emptyset \oplus R$$

□

Lema 6.23. *A operação* $\oplus$ *é comutativa.*

Demonstração. Pela definição 6.17, se $R_1 \oplus R_2$ é definido, então $\text{fnames}(R_1) \cap \text{fnames}(R_2) = \emptyset$ e $R_1 \oplus R_2 = R_1 \cup R_2$. Como $\cap$ são $\cup$ operações comutativas, temos que $\text{fnames}(R_2) \cap \text{fnames}(R_1) = \emptyset$ e que $R_1 \oplus R_2 = R_2 \cup R_1 = R_2 \oplus R_1$. □

Lema 6.24. *Se* $R_1 \ominus R_2$ *é definido então* $\text{fnames}(R_1 \ominus R_2) = \text{fnames}(R_1) - \text{fnames}(R_2)$.

Demonstração. Se $R_1 \ominus R_2$ é definido então $R_2 \subseteq R_1$, e

$$\begin{aligned}
\text{fnames}(R_1 \ominus R_2) &= \text{fnames}(R_1 - R_2) \\
&= \{f \mid (f,\tau) \in R_1 - R_2, \text{ para algum } \tau\} \\
&= \{f \mid (f,\tau) \in R_1 \text{ e } (f,\tau) \notin R_2, \text{ para algum } \tau\}
\end{aligned}$$

Pelo lema 6.10, se $R_2 \subseteq R_1$, $(f,\tau) \in R_1$ e $(f,\tau) \notin R_2$, então $f \notin \text{fnames}(R_2)$. Portanto:

$$\begin{aligned}
\text{fnames}(R_1 \ominus R_2) &= \{f \mid (f,\tau) \in R_1 \text{ e } (f,\tau) \notin R_2, \text{ para algum } \tau\} \\
&= \{f \mid (f,\tau) \in R_1, \text{ para algum } \tau, \text{ e } f \notin \text{fnames}(R_2)\} \\
&= \{f \mid f \in \text{fnames}(R_1) \text{ e } f \notin \text{fnames}(R_2)\} \\
&= \text{fnames}(R_1) - \text{fnames}(R_2)
\end{aligned}$$

□

Lema 6.25. *Se $R_1 \oplus R_2$ é definido então* $\text{fnames}(R_1 \oplus R_2) = \text{fnames}(R_1) \cup \text{fnames}(R_2)$

Demonstração. Se $R_1 \oplus R_2$ é definido então:

$$\begin{aligned} \text{fnames}(R_1 \oplus R_2) &= \text{fnames}(R_1 \cup R_2) \\ &= \text{fnames}(R_1) \cup \text{fnames}(R_2) \text{ (por 6.7)} \end{aligned}$$

□

Lema 6.26. *O operador $\oplus$ é associativo*

Demonstração. Pela definição 6.17, se $(R_1 \oplus R_2) \oplus R_3$ está definido, então as seguintes condições devem ser verdades:

$$\begin{aligned} \text{fnames}(R_1) \cap \text{fnames}(R_2) &= \emptyset, \text{ para que } R_1 \oplus R_2 \text{ seja definido, e} \\ \text{fnames}(R_1 \oplus R_2) \cap \text{fnames}(R_3) &= \emptyset, \text{ para que } (R_1 \oplus R_2) \oplus R_3 \text{ seja definido.} \end{aligned}$$

Pelo lema 6.25, temos:

$$\begin{aligned} \emptyset &= \text{fnames}(R_1 \oplus R_2) \cap \text{fnames}(R_3) \\ &= (\text{fnames}(R_1) \cup \text{fnames}(R_2)) \cap \text{fnames}(R_3) \\ &= (\text{fnames}(R_1) \cap \text{fnames}(R_3)) \cup (\text{fnames}(R_2) \cap \text{fnames}(R_3)) \end{aligned}$$

Como $X \cup Y = \emptyset$ implica que $X = \emptyset$ e $Y = \emptyset$, temos:

$$\begin{aligned} \text{fnames}(R_1) \cap \text{fnames}(R_3) &= \emptyset \\ \text{fnames}(R_2) \cap \text{fnames}(R_3) &= \emptyset \end{aligned}$$

Portanto, $R_2 \oplus R_3$ é definido.
Substituindo fnames(X) por fn(X), temos:

$$\begin{aligned} \text{fn}(R_1) \cap \text{fn}(R_2 \oplus R_3) &= \text{fn}(R_1) \cap (\text{fn}(R_2) \cup \text{fn}(R_3)) \\ &= (\text{fn}(R_1) \cap \text{fn}(R_2)) \cup (\text{fn}(R_1) \cap \text{fn}(R_3)) \\ &= \emptyset \cup \emptyset = \emptyset \end{aligned}$$

Portanto, $R_1 \oplus (R_2 \oplus R_3)$ é definido.
Podemos concluir que, se $(R_1 \oplus R_2) \oplus R_3$ é definido, então $R_1 \oplus (R_2 \oplus R_3)$ também é definido.

Por outro lado, suponha que $R_1 \oplus (R_2 \oplus R_3)$ seja definido. Pela definição 6.17 e pelo lema 6.25, o seguinte deve ser verdade:

- $\text{fnames}(R_2) \cap \text{fnames}(R_3) = \emptyset$;
- $\text{fnames}(R_1) \cap (\text{fnames}(R_2) \cup \text{fnames}(R_3)) = \emptyset$.

De $\text{fnames}(R_1) \cap (\text{fnames}(R_2) \cup \text{fnames}(R_3)) = \emptyset$ podemos deduzir que $\text{fnames}(R_1) \cap \text{fnames}(R_2) = \emptyset$ e $\text{fnames}(R_1) \cap \text{fnames}(R_3) = \emptyset$ e, portanto, que $R_1 \oplus R_2$ é definido.

Também temos:

$$\begin{array}{ll} (\text{fnames}(R_1) \cup \text{fnames}(R_2)) \cap \text{fnames}(R_3) & = \\ (\text{fnames}(R_1) \cap \text{fnames}(R_3)) \cup (\text{fnames}(R_2) \cap \text{fnames}(R_3)) & = \emptyset \end{array}$$

Podemos concluir, então, que se $R_1 \oplus (R_2 \oplus R_3)$ é definido, então $(R_1 \oplus R_2) \oplus R_3$ também é.

Seguimos, então, com a demonstração da igualdade:

$$\begin{array}{rcl} R_1 \oplus (R_2 \oplus R_3) & = & R_1 \oplus (R_2 \cup R_3) \\ & = & R_1 \cup (R_2 \cup R_3) \\ & = & (R_1 \cup R_2) \cup R_3 \\ & = & (R_1 \oplus R_2) \cup R_3 \\ & & (R_1 \oplus R_2) \oplus R_3 \end{array}$$

□

Lema 6.27. *O conjunto vazio é uma identidade à direita de* $\ominus$.

Demonstração. Como $\emptyset \subseteq R$, para qualquer R, então $R \ominus \emptyset$ está definido para qualquer R e é igual a:

$$R \ominus \emptyset = R - \emptyset = R$$

□

Lema 6.28. *Se* $R_1 \oplus R_2$ *está definido, então* $(R_1 \oplus R_2) \ominus R_2 = R_1$.

Demonstração. Se $R_1 \oplus R_2$ está definido, então, pela definição 6.17, temos:

$$\begin{array}{l} \text{fnames}(R_1) \cap \text{fnames}(R_2) = \emptyset \text{ e} \\ R_1 \oplus R_2 = R_1 \cup R_2 \end{array}$$

Como $R_2 \subseteq R_1 \cup R_2 \subseteq R_1 \oplus R_2$, então, pela definição 6.18, $(R_1 \oplus R_2) \ominus R_2$ é definido e é igual a:

$$\begin{array}{rcl} (R_1 \oplus R_2) \ominus R_2 & = & (R_1 \oplus R_2) - R_2 \\ & = & (R_1 \cup R_2) - R_2 \\ & = & (R_1 - R_2) \cup (R_2 - R_2) \\ & = & R_1 - R_2 \\ & = & R_1 - (R_1 \cap R_2) \end{array}$$

Pelo lema 6.11, temos que $R_1 \cap R_2 = \emptyset$ e, portanto, que:

$$(R_1 \oplus R_2) \ominus R_2 = R_1$$

□

Lema 6.29. *Se $R_1 \ominus R_2$ é definido, então $(R_1 \ominus R_2) \oplus R_2 = R_1$.*

Demonstração. Se $R_1 \ominus R_2$ é definido, então, pela definição 6.18, temos:

$$\begin{aligned} &R_2 \subseteq R_1 \text{ e} \\ &R_1 \ominus R_2 = R_1 - R_2 \end{aligned}$$

Pelo lema 6.24, temos:

$$\begin{aligned} \text{fnames}(R_1 \ominus R_2) \cap \text{fnames}(R_2) &= \text{fnames}(R_1 - R_2) \cap \text{fnames}(R_2) \\ &= (\text{fnames}(R_1) - \text{fnames}(R_2)) \cap \text{fnames}(R_2) \\ &= (\text{fnames}(R_1) \cap \text{fnames}(R_2)) - \text{fnames}(R_2) \\ &= \emptyset \end{aligned}$$

Portanto, pela definição 6.17, $(R_1 \ominus R_2) \oplus R_2$ é definido e é igual a:

$$\begin{aligned} (R_1 \ominus R_2) \oplus R_2 &= (R_1 \ominus R_2) \cup R_2 \\ &= (R_1 - R_2) \cup R_2 \\ &= R_1 \cup R_2 \end{aligned}$$

Porém, como $R_2 \subseteq R_1$, então:

$$(R_1 \ominus R_2) \oplus R_2 = R_1$$

□

Lema 6.30. $(R_1 \ominus R_2) \ominus R_3 = R_1 \ominus (R_2 \oplus R_3)$.

Demonstração. Para provar esta igualdade, temos que provar que se o lado direito é definido, então o lado esquerdo também o é, e vice-versa, e depois provar que quando definidos, os dois lados têm o mesmo valor.

Primeiramente, provaremos que se $(R_1 \ominus R_2) \ominus R_3$ é definido então $R_1 \ominus (R_2 \oplus R_3)$ também o é.

Para que $R_1 \ominus R_2$ seja definido, é necessário que $R_2 \subseteq R_1$. Para que $(R_1 \ominus R_2) \ominus R_3$ seja definido, é necessário que $R_3 \subseteq R_1 \oplus R_2$, o que significa que $R_3 \subseteq R_1 - R_2$, pela definição de $\ominus$.

$R_3 \subseteq R_1 - R_2$ implica que $R_3 \subseteq R_1$ e que $R_3 \cap R_2 = \emptyset$. Pelo lema 6.12, temos que $\text{fnames}(R_2) \cap \text{fnames}(R_3) = \emptyset$, o que nos permite concluir que $R_2 \oplus R_3$ é definido.

Por outro lado, se $R_1 \ominus (R_2 \oplus R_3)$ é definido, então $R_2 \oplus R_3 \subseteq R_1$ e $\text{fnames}(R_2) \cap \text{fnames}(R_3) = \emptyset$, implicando que $R_2 \cup R_3 \subseteq R_1$. Pelo lema 6.11, temos que $R_2 \cap R_3 = \emptyset$.

$$\begin{aligned}
R_2 \oplus R_3 &\subseteq R_1 \\
R_2 \cup R_3 &\subseteq R_1 \\
(R_2 \cup R_3) - R_2 &\subseteq R_1 \\
(R_2 - R_2) \cup (R_3 - R_2) &\subseteq R_1 - R_2 \\
R_3 - R_2 &\subseteq R_1 - R_2 \\
(R_3 - R_2) \cap R_3 &\subseteq (R_1 - R_2) \cap R_3 \\
(R_3 \cap R_3) - (R_2 \cap R_3) &\subseteq (R_1 - R_2) \cap R_3 \\
R_3 &\subseteq (R_1 - R_2) \cap R_3 \\
R_3 &\subseteq R_1 - R_2
\end{aligned}$$

Como $R_2 \subseteq R_1$, então $R_1 \ominus R_2$ é definido e é igual a $R_1 - R_2$ e como $R_3 \subseteq R_1 - R_2$, então $(R_1 \ominus R_2) \ominus R_3$ também é definido.

Pela definição de $\oplus$ e $\ominus$, temos:

$$\begin{aligned}
R_1 \ominus (R_2 \oplus R_3) &= R_1 \ominus (R_2 \cup R_3) \\
&= R_1 - (R_2 \cup R_3) \\
&= (R_1 - R_2) - R_3 \\
&= (R_1 \ominus R_2) \ominus R_3
\end{aligned}$$

□

Lema 6.31. *Se* $(R_1 \oplus R_2) \ominus R_3$ *é definido, então* $\text{cortypes}(R_2, R_3) \subseteq \Delta\mathbf{T}$.

Demonstração. Suponha que $(R_1 \oplus R_2) \ominus R_3$ seja definido, mas que $\text{cortypes}(R_2, R_3) \not\subseteq \Delta\mathbf{T}$. Deve haver então f, τ e τ' tais que $(f, \tau) \in R_2$, $(f, \tau') \in R_3$ e $\tau \neq \tau'$.

Pela definição 6.18, $R_3 \subseteq R_1 \oplus R_2$, o que significa que $(f, \tau') \in R_1 \oplus R_2$. Pela definição 6.17, temos que $(f, \tau') \in R_1 \cup R_2$ e $\text{fnames}(R_1) \cap \text{fnames}(R_2) = \emptyset$. Nesta situação, alguma das condições abaixo precisa ser verdade:

- $(f, \tau') \in R_2$: neste caso, $\{(f, \tau), (f, \tau')\} \subseteq R_2$, o que significa que $\tau = \tau'$, o que é uma contradição;
- $(f, \tau') \in R_1$: neste caso, temos que $f \in \text{fnames}(R_1)$ e $f \in \text{fnames}(R_2)$, o que implica que $\text{fnames}(R_1) \cap \text{fnames}(R_2) \neq \emptyset$, o que também é uma contradição.

□

Lema 6.32. *Se* $((R_1 \ominus R_2) \oplus R_3) \ominus R_4$ *é definido, então* $R_2 \cup (R_4 - R_3)$ *é um conjunto de campos.*

Demonstração. Suponha que $((R_1 \ominus R_2) \oplus R_3) \ominus R_4$ é definido, mas que $R_2 \cup (R_4 - R_3)$ não seja um conjunto de campos.

Se $((R_1 \ominus R_2) \oplus R_3) \ominus R_4$ é definido, então:

- $R_4 \subseteq (R_1 \ominus R_2) \oplus R_3$;
- $\text{fnames}(R_1 \ominus R_2) \cap \text{fnames}(R_3) = \emptyset$;
- $R_2 \subseteq R_1$.

Se $R_2 \cup (R_4 - R_3)$ não é um conjunto de campos, então deve haver f, τ e τ' tais que $\{(f, \tau), (f, \tau')\} \subseteq R_2 \cup (R_4 - R_3)$ e $\tau \neq \tau'$. Neste caso, uma das seguintes condições deve ser verdade:

- $\{(f, \tau), (f, \tau')\} \subseteq R_2$: pela definição 6.1, $\tau = \tau'$, o que é uma contradição;
- $\{(f, \tau), (f, \tau')\} \subseteq R_4 - R_3$: pelo lema 6.16, sabemos que $R_4 - R_3$ é um conjunto de campos e, portanto, pela definição 6.1, $\tau = \tau'$, o que é uma contradição;
- $(f, \tau) \in R_2$ e $(f, \tau') \in R_4 - R_3$, ou $(f, \tau') \in R_2$ e $(f, \tau) \in R_4 - R_3$.

Suponha, sem perda de generalidade, que $(f, \tau) \in R_2$ e $(f, \tau') \in R_4 - R_3$. Como $R_4 \subseteq (R_1 \ominus R_2) \oplus R_3$, então $(f, \tau') \in (R_1 \ominus R_2) \oplus R_3$. Porém, temos que $(f, \tau') \notin R_3$, porque $(f, \tau') \in R_4 - R_3$. Deve ser o caso, então, que $(f, \tau') \in R_1 \ominus R_2 = R_1 - R_2$. Pela definição 6.1, teríamos então que $\tau = \tau'$, o que é uma contradição. □

Lema 6.33. *Se $((R_1 \ominus R_2) \oplus R_3) \ominus R_4$ é definido, então $(R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus (R_3 - R_4)$ é definido.*

Demonstração. Se $(R_1 \ominus R_2) \oplus R_3$ é definido, então $R_2 \subseteq R_1$, $\text{fnames}(R_1 \ominus R_2) \cap \text{fnames}(R_3) = \emptyset$, e $(R_1 \ominus R_2) \oplus R_3 = (R_1 - R_2) \cup R_3$. Pelo lema 6.11, teríamos também que $(R_1 - R_2) \cap R_3 = \emptyset$.

Se $((R_1 \ominus R_2) \oplus R_3) \ominus R_4$ é definido, então $R_4 \subseteq (R_1 - R_2) \cup R_3$ e, pelo lema 6.32, $R_2 \cup (R_4 - R_3)$ é um conjunto de campos.

Temos então que:

$$\begin{aligned} R_4 &\subseteq (R_1 - R_2) \cup R_3 \\ R_4 - R_3 &\subseteq (R_1 - R_2) - R_3 \\ R_4 - R_3 &\subseteq R_1 - R_2 \text{ (pois } (R_1 - R_2) \cap R_3 = \emptyset) \\ R_2 \cup (R_4 - R_3) &\subseteq R_1 \cup R_2 \\ R_2 \cup (R_4 - R_3) &\subseteq R_1 \text{ (pois } R_2 \subseteq R_1) \end{aligned}$$

Isto significa que $R_1 \ominus (R_2 \cup (R_4 - R_3))$ é definido. Além disso, pelo lema 6.31, temos que $\text{cortypes}(R_3, R_4) \subseteq \Delta\mathbf{T}$ e, pelo lema 6.14, que $\text{fnames}(R_3 - R_4) = \text{fnames}(R_3) - \text{fnames}(R_4)$.

Nas seguintes derivações, a função f é usada como abreviação de fnames, por causa do tamanho das expressões:

$$\begin{array}{lll}
\mathrm{f}(R_1 \ominus (R_2 \cup (R_4 - R_3)) & = & \\
\mathrm{f}(R_1) - \mathrm{f}(R_2 \cup (R_4 - R_3)) & = & (6.7) \\
\mathrm{f}(R_1) - (\mathrm{f}(R_2) \cup \mathrm{f}(R_4 - R_3)) & = & (6.14) \\
\mathrm{f}(R_1) - (\mathrm{f}(R_2) \cup (\mathrm{f}(R_4) - \mathrm{f}(R_3))) & = & \\
(\mathrm{f}(R_1) - \mathrm{f}(R_2)) - (\mathrm{f}(R_4) - \mathrm{f}(R_3)) & = & \\
((\mathrm{f}(R_1) - \mathrm{f}(R_2)) - \mathrm{f}(R_4)) \cup ((\mathrm{f}(R_1) - \mathrm{f}(R_2)) \cap \mathrm{f}(R_3)) & = & \\
(\mathrm{f}(R_1) - (\mathrm{f}(R_2) \cup \mathrm{f}(R_4))) \cup ((\mathrm{f}(R_1) \cap \mathrm{f}(R_3)) - \mathrm{f}(R_2)) & &
\end{array}$$

e

$$\begin{array}{ll}
(\mathrm{f}(R_1 \ominus (R_2 \cup (R_4 - R_3))) \cap \mathrm{f}(R_3) & = \\
((\mathrm{f}(R_1) - (\mathrm{f}(R_2) \cup \mathrm{f}(R_4))) \cup ((\mathrm{f}(R_1) \cap \mathrm{f}(R_3)) - \mathrm{f}(R_2))) \cap \mathrm{f}(R_3) & = \\
((\mathrm{f}(R_1) - (\mathrm{f}(R_2) \cup \mathrm{f}(R_4))) \cap \mathrm{f}(R_3)) \cup (((\mathrm{f}(R_1) \cap \mathrm{f}(R_3)) - \mathrm{f}(R_2)) \cap \mathrm{f}(R_3)) & = \\
((\mathrm{f}(R_1) \cap \mathrm{f}(R_3)) - (\mathrm{f}(R_2) \cup \mathrm{f}(R_4))) \cup ((\mathrm{f}(R_1) \cap \mathrm{f}(R_3)) - \mathrm{f}(R_2)) & = \\
((\mathrm{f}(R_1) \cap \mathrm{f}(R_3)) - \mathrm{f}(R_2) & = \\
(((\mathrm{f}(R_1) - \mathrm{f}(R_2)) \cap \mathrm{f}(R_3)) - (\mathrm{f}(R_2) \cap \mathrm{f}(\mathrm{R}_3)) & = \\
\emptyset - (\mathrm{f}(R_2) \cap \mathrm{f}(R_3)) & = \\
\emptyset & =
\end{array}$$

e

$$\begin{array}{lll}
(\mathrm{f}(R_1 \ominus (R_2 \cup (R_4 - R_3))) \cap (\mathrm{f}(R_3 - R_4)) & = & (6.14) \\
(\mathrm{f}(R_1 \ominus (R_2 \cup (R_4 - R_3))) \cap (\mathrm{f}(R_3) - \mathrm{f}(R_4)) & = & \\
((\mathrm{f}(R_1 \ominus (R_2 \cup (R_4 - R_3))) \cap \mathrm{f}(R_3)) - ((\mathrm{f}(R_1 \ominus (R_2 \cup (R_4 - R_3))) \cap \mathrm{f}(R_4)) & = & \\
\emptyset - ((\mathrm{f}(R_1 \ominus (R_2 \cup (R_4 - R_3))) \cap \mathrm{f}(R_4)) & = & \\
\emptyset & &
\end{array}$$

Portanto, $(R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus (R_3 - R_4)$ é definido.

□

Lema 6.34. *Se* $((R_1 \ominus R_2) \oplus R_3) \ominus R_4$ *é definido, então* $\mathrm{cortypes}(\mathrm{R}_3, \mathrm{R}_4) \subseteq \Delta\mathbf{T}$.

Demonstração. Isto é uma consequência direta do lema 6.31. □

Lema 6.35. *Se* $((R_1 \ominus R_2) \oplus R_3) \ominus R_4$ *é definido, então* $(\mathrm{fnames}(R_2) - \mathrm{fnames}(R_3)) \cap \mathrm{fnames}(R_4) = \emptyset$.

Demonstração. Suponha que $((R_1 \ominus R_2) \oplus R_3) \ominus R_4$ seja definido, mas que $(\mathrm{fnames}(R_2) - \mathrm{fnames}(R_3)) \cap \mathrm{fnames}(R_4) \neq \emptyset$. Deve haver, então, um f tal que $f \in \mathrm{fnames}(R_2) - \mathrm{fnames}(R_3)$ e $f \in \mathrm{fnames}(R_4)$, o que implica que $f \in \mathrm{fnames}(R_2)$ e que $f \notin \mathrm{fnames}(R_3)$.

Deve haver também um τ tal que $(f, \tau) \in R_2$. Porém, como $R_2 \subseteq R_1$, então $(f, \tau) \in R_1$. Neste caso, não poderia haver um τ' tal que $(f, \tau') \in R_1 \ominus R_2$. Como também não existe um τ' tal que $(f, \tau') \in \mathrm{fnames}(R_3)$, então não existe um τ' tal que $(f, \tau') \in (R_1 \ominus R_2) \oplus R_3$.

Entretanto, como é necessário que exista um τ' tal que $(f, \tau') \in R_4$, e como $R_4 \subseteq (R_1 \ominus R_2) \oplus R_3$, teríamos então que $(f, \tau') \in (R_1 \ominus R_2) \oplus R_3$, o que seria uma contradição. □

Lema 6.36. *Se* $(R_1 \ominus R_2) \oplus R_3$ *e* $(R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus (R_3 - R_4)$ *são definidos,* $(\text{fnames}(R_2) - \text{fnames}(R_3)) \cap \text{fnames}(R_4) = \emptyset$ *e* $\text{cortypes}(R_3, R_4) \subseteq \Delta\mathbf{T}$*, então* $((R_1 \ominus R_2) \oplus R_3) \ominus R_4$ *é definido.*

Demonstração. Suponha que $(R_1 \ominus R_2) \oplus R_3$, $(R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus (R_3 - R_4)$, $(\text{fnames}(R_2) - \text{fnames}(R_3)) \cap \text{fnames}(R_4) = \emptyset$ e $\text{cortypes}(R_3, R_4) \subseteq \Delta\mathbf{T}$, mas que $((R_1 \ominus R_2) \oplus R_3) \ominus R_4$ seja indefinido.

Como $(R_1 \ominus R_2) \oplus R_3$ é definido, para que $((R_1 \ominus R_2) \oplus R_3) \ominus R_4$ seja indefinido, é preciso que $R_4 \not\subseteq (R_1 \ominus R_2) \oplus R_3$, o que implica que deve haver um f e um τ tais que $(f, \tau) \in R_4$ e $(f, \tau) \notin (R_1 \ominus R_2) \oplus R_3$. Para que esta última proposição seja verdade, uma das seguintes condições deve ser verdade também:

Condição 6.37. Não existe um τ' tal que $(f, \tau') \in (R_1 \ominus R_2) \oplus R_3$.

Condição 6.38. Existe um τ' tal que $(f, \tau') \in (R_1 \ominus R_2) \oplus R_3$. Porém, $\tau \neq \tau'$.

Para que a condição 6.37 seja verdade, uma das seguintes condições deve ser verdade:

Condição 6.39. Não existe um τ' tal que $(f, \tau') \in R_3$, ou que $(f, \tau') \in R_1$, ou que $(f, \tau') \in R_2$.

Condição 6.40. Não existe um τ' tal que $(f, \tau') \in R_3$, mas existe um τ' tal que $(f, \tau') \in R_1$ e que $(f, \tau') \in R_2$.

Como $(R_1 \ominus R_2) \oplus R_3$ é definido, o que implica que $\text{fnames}(R_1 \ominus R_2) \cap \text{fnames}(R_3) = \emptyset$, para que a condição 6.38 seja verdade, uma das seguintes condições deve ser verdade:

Condição 6.41. $(f, \tau') \in R_1 \ominus R_2$ e não existe τ'' tal que $(f, \tau'') \in R_3$.

Condição 6.42. $(f, \tau') \in R_3$ e não existe τ'' tal que $(f, \tau'') \in R_1 \ominus R_2$.

Para demonstrar o lema, é suficiente provar que as condições 6.39, 6.40, 6.41 e 6.42 levam a contradições:

Suponha que a condição 6.39 seja verdade: não existe um τ' tal que $(f, \tau') \in R_3$, ou que $(f, \tau') \in R_1$, ou que $(f, \tau') \in R_2$. Como $(f, \tau) \in R_4$, então $(f, \tau) \in R_4 - R_3$ e $(f, \tau) \in R_2 \cup (R_4 - R_3)$.

Uma das hipóteses é a de que $(R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus (R_3 - R_4)$ seja definido, o que implica que $R_1 \ominus (R_2 \cup (R_4 - R_3))$ é definido e que $R_2 \cup (R_4 - R_3) \subseteq R_1$. Como $(f, \tau) \in R_2 \cup (R_4 - R_3)$, teríamos então que $(f, \tau) \in R_1$, o que contradiz a condição 6.39.

Suponha que a condição 6.40 seja verdadeira: não existe um τ' tal que $(f, \tau') \in R_3$, mas existe um τ' tal que $(f, \tau') \in R_1$ e que $(f, \tau') \in R_2$. Teríamos então que $f \in \text{fnames}(R_2)$, $f \notin \text{fnames}(R_3)$ e que $f \in \text{fnames}(R_4)$, o que implica que $f \in (\text{fnames}(R_2) - \text{fnames}(R_3)) \cap \text{fnames}(R_4)$, que contradiz a hipótese do lema.

Suponha que a condição 6.41 seja verdade: $\tau' \neq \tau$, $(f, \tau') \in R_1 \ominus R_2$ e não existe um τ'' tal que $(f, \tau'') \in R_3$. Como $f \notin \text{fnames}(R_3)$, então $(f, \tau) \in R_4 - R_3$ e $(f, \tau) \in R_2 \cup (R_4 - R_3)$. Supomos que $R_1 \ominus (R_2 \cup (R_4 - R_3))$ é definido, o que implica que $(f, \tau) \in R_1$. Também

temos que $(f, \tau') \in R_1 \ominus R_2$ implica que $(f, \tau') \in R_1$. Juntando as duas partes, temos que $\{(f, \tau), (f, \tau')\} \in R_1$, o que implica, pela definição 6.1, que $\tau = \tau'$, o que contradiz a condição.

Finalmente, suponha que a condição 6.42 seja verdade: $\tau \neq \tau'$, $(f, \tau') \in R_3$ e não existe um τ'' tal que $(f, \tau'') \in R_1 \ominus R_2$. Como $(f, \tau') \in R_3$ e $(f, \tau) \in R_4$, então $(\tau, \tau') \in \text{cortypes}(R_3, R_4)$. Mas como $\text{cortypes}(R_3, R_4) \subseteq \Delta\mathbf{T}$, então $\tau = \tau'$, o que contradiz a condição. □

Lema 6.43. *Se $(R_1 \ominus R_2) \oplus R_3$ é definido, então a seguinte proposição é verdadeira:*
$((R_1 \ominus R_2) \oplus R_3) \ominus R_4$ é definido se e somente se: $(R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus (R_3 - R_4)$ é definido; $\text{cortypes}(R_3, R_4) \subseteq \Delta\mathbf{T}$*; e* $(\text{fnames}(R_2) - \text{fnames}(R_3)) \cap \text{fnames}(R_4) = \emptyset$.

Demonstração. A prova da implicação na direção esquerda-direita é uma consequência direta dos lemas 6.33, 6.34 e 6.35. A prova da implicação na direção inversa é dada pelo lema 6.36. □

Lema 6.44. *Se $((R_1 \ominus R_2) \oplus R_3) \ominus R_4$ e $(R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus (R_3 - R_4)$ é definido, então:*

$$((R_1 \ominus R_2) \oplus R_3) \ominus R_4 \quad = \quad (R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus (R_3 - R_4)$$

Demonstração. Pela definição 6.18 sabemos que $R_4 \subseteq (R_1 - R_2) \cup R_3$. Temos portanto que:

$$R_4 = (R_4 \cap R_3) \cup (R_4 - R_3)$$

e que:

$$\begin{aligned}(R_1 - R_2) - R_4 &= (R_1 - R_2) - ((R_4 - R_3) \cup (R_4 \cap R_3)) \\ &= ((R_1 - R_2) - (R_4 \cap R_3)) - (R_4 - R_3)\end{aligned}$$

Pela definição 6.17, temos que $\text{fnames}(R_1 \ominus R_2) \cap \text{fnames}(R_3) = \emptyset$, o que implica, pelo lema 6.11, que $(R_1 \ominus R_2) \cap R_3 = \emptyset$, ou, pela definição 6.18, que $(R_1 - R_2) \cap R_3 = \emptyset$, ou ainda que $(R_1 - R_2) \cap (R_3 \cap R_4) = \emptyset$. Podemos concluir então que $(R_1 - R_2) - (R_4 \cap R_3) = R_1 - R_2$:

$$\begin{aligned}(R_1 - R_2) - R_4 &= ((R_1 - R_2) - (R_4 \cap R_3)) - (R_4 - R_3) \\ &= (R_1 - R_2) - (R_4 - R_3) \\ &= (R_1 - (R_2 \cup (R_4 - R_3)))\end{aligned}$$

Pelas definições 6.17 e 6.18, e tomando como hipótese que $(R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus (R_3 - R_4)$ é definido, temos:

$$\begin{aligned}((R_1 \ominus R_2) \oplus R_3) \ominus R_4 &= ((R_1 - R_2) \cup R_3) - R_4 \\ &= ((R_1 - R_2) - R_4) \cup (R_3 - R_4) \\ &= (R_1 - (R_2 \cup (R_4 - R_3))) \cup (R_3 - R_4) \\ &= (R_1 \ominus (R_2 \cup (R_4 - R_3))) \cup (R_3 - R_4) \\ &= (R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus (R_3 - R_4)\end{aligned}$$

□

Lema 6.45. *Se* $(((R_1 \ominus R_2) \oplus R_3) \ominus R_4) \oplus R_5$ *é definido, então* $(R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus ((R_3 - R_4) \oplus R_5)$ *também é definido, e o seguinte é verdade:*

$$(((R_1 \ominus R_2) \oplus R_3) \ominus R_4) \oplus R_5 = ((R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus ((R_3 - R_4) \oplus R_5).$$

Demonstração. Pelo lema 6.43, como $((R_1 \ominus R_2) \oplus R_3) \ominus R_4$ é definido, então $((R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus ((R_3 - R_4) \oplus R_5)$ também é definido.

Temos também que, pelo lema 6.44, o seguinte é verdade:

$$((R_1 \ominus R_2) \oplus R_3) \ominus R_4 = (R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus (R_3 - R_4)$$

Portanto, temos:

$$(((R_1 \ominus R_2) \oplus R_3) \ominus R_4) \oplus R_5 \quad = \quad ((R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus (R_3 - R_4)) \oplus R_5$$

Pelo lema 6.26, temos:

$$\begin{aligned}(((R_1 \ominus R_2) \oplus R_3) \ominus R_4) \oplus R_5 &= ((R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus (R_3 - R_4)) \oplus R_5 \\ &= ((R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus ((R_3 - R_4) \oplus R_5)\end{aligned}$$

□

Lema 6.46. *Se* $((R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus ((R_3 - R_4) \oplus R_5)$ *é definido,* $\text{cortypes}(R_3, R_4) \subseteq \Delta\mathbf{T}$ *e* $(\text{fnames}(R_2) - \text{fnames}(R_3)) \cap \text{fnames}(R_4) = \emptyset$*, então* $(((R_1 \ominus R_2) \oplus R_3) \ominus R_4) \oplus R_5$ *é definido, e o seguinte é verdade:*

$$((R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus ((R_3 - R_4) \oplus R_5) = (((R_1 \ominus R_2) \oplus R_3) \ominus R_4) \oplus R_5$$

.

Demonstração. Pelo lema 6.26, se

$$((R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus ((R_3 - R_4) \oplus R_5)$$

é definido, então

$$(((R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus (R_3 - R_4)) \oplus R_5$$

também é definido.

Como $((R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus (R_3 - R_4)$ é definido, $\text{cortypes}(R_3, R_4) \subseteq \Delta\mathbf{T}$ e $(\text{fnames}(R_2) - \text{fnames}(R_3)) \cap \text{fnames}(R_4) = \emptyset$, então, pelo lema 6.36, temos que $((R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus (R_3 - R_4)$ é definido, e pelo lema 6.44 temos:

$$\begin{aligned}&((R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus ((R_3 - R_4) \oplus R_5) \quad = \\ &(((R_1 \ominus (R_2 \cup (R_4 - R_3))) \oplus (R_3 - R_4)) \oplus R_5 \quad = \\ &(((R_1 \ominus R_2) \oplus R_3) \ominus R_4) \oplus R_5\end{aligned}$$

□

6.3 Termos de conjuntos de campos

Definição 6.47. O conjunto de termos de conjuntos de campos é o menor conjunto que satisfaz as seguintes condições:

- Um conjunto de campos é um termo de conjunto de campos;
- Se α é uma variável de tipo, então r_α é um termo de conjunto de campos. Tais termos são chamados variáveis de conjuntos de campos;
- Se t_1 e t_2 são termos, então $t_1 \oplus t_2$ é um termo;
- Se t_1e t_2 são termos, então $t_1 \ominus t_2$ é um termo.

Definição 6.48. O tamanho de um termo t é o número de subtermos de t:

$$\text{size}(t) = \begin{cases} 1 & \text{se } t = R \text{ ou } t = r_\alpha \\ \text{size}(t_1) + \text{size}(t_2) + 1 & \text{se } t = t_1 \oplus t_2 \text{ ou } t = t_1 \ominus t_2 \end{cases}$$

Uma propriedade útil da função size é a de que se t_1é um subtermo próprio de t_2, então $\text{size}(t_1) < \text{size}(t_2)$.

Definição 6.49. O conjunto de variáveis de conjuntos de campos presentes em um termo t é dado pela função fv:

$$\text{fv}(t) = \begin{cases} \{r_\alpha\} & \text{se } t = r_\alpha \\ \text{fv}(t_1) \cup \text{fv}(t_2) & \text{se } t = t_1 \text{ op } t_2, \text{ onde op} \in \{\oplus, \ominus\} \\ \emptyset & \text{se } t = R \end{cases}$$

Definição 6.50. Um termo t é chamado um termo *ground* se $\text{fv}(t) = \emptyset$.

Todo termo *ground* corresponde a uma expressão envolvendo conjuntos de campos e os operadores $\oplus$ e $\ominus$. No entanto nem todas as expressões denotam valores definidos. Como $\oplus$ e $\ominus$ são operadores parciais, algumas condições são necessárias para que um termo *ground* denote um conjunto de campos definido.

Definição 6.51. Um termo *ground* é chamado um *termo válido* se ele denota um valor definido. O valor denotado por um termo válido t é representado por $\text{val}(t)$.

Se um termo é *ground*, mas não representa um valor definido, então é chamado *termo inválido*.

6.4 Substituições

Uma substituição S sobre tipos simples pode ser estendida para um substituição sobre conjuntos de campos e sobre termos de conjuntos de campos, como mostram as seguintes definições:

Definição 6.52. A aplicação de uma substituição S a um conjunto de campos R é definida como:

$$S(R) = \{(f, S\tau) \mid (f, \tau) \in R\}$$

Como a substituição só altera os tipos dos campos, e não os nomes dos mesmos, o conjunto resultante é um conjunto de campos, conforme a definição 6.1.

Lema 6.53. *A aplicação de substituição distribui sobre a função de representação* $[\![\ ]\!]$*, quando restrita a tipos registro.*

Demonstração. Se τ é um tipo registro $\{\mathrm{f}_1 : \tau_1, \ldots, \mathrm{f}_n : \tau_n\}$ então, pelas definições 6.2 e 6.52, temos:

$$\begin{aligned} S([\![\tau]\!]) &= S(\{(\mathrm{f}_1, \tau_1), \ldots, (\mathrm{f}_n, \tau_n)\}) \\ &= \{(\mathrm{f}_1, S\tau_1), \ldots, (\mathrm{f}_n, S\tau_n)\} \\ &= [\![S\tau]\!] \end{aligned}$$

□

Lema 6.54. *A aplicação de substituição distribui sobre a inversa da função de representação (* $[\![\ ]\!]^{-1}$*).*

Demonstração. Se R é um conjunto de campos $\{(\mathrm{f}_1, \tau_1), \ldots, (\mathrm{f}_n, \tau_n)\}$ então, pelas definições 6.2 e 6.52, temos:

$$\begin{aligned} S([\![R]\!]^{-1}) &= S([\![\{(\mathrm{f}_1, \tau_1), \ldots, (\mathrm{f}_n, \tau_n)\}]\!]^{-1}) \\ &= S(\{\mathrm{f}_1 : \tau_1, \ \ldots, \ \mathrm{f}_n : \tau_n\}) \\ &= \{\mathrm{f}_1 : S\tau_1, \ \ldots, \ \mathrm{f}_n : S\tau_n\} \\ &= [\![\{(\mathrm{f}_1, S\tau_1), \ \ldots, \ (\mathrm{f}_n, S\tau_n)\}]\!]^{-1} \\ &= [\![S(R)]\!]^{-1} \end{aligned}$$

□

Lema 6.55. *Se R_1 e R_2 são conjuntos de campos, onde $R_1 = [\![\tau_1]\!]$ e $R_2 = [\![\tau_2]\!]$, e se S é uma substituição, então $S\tau_1 = S\tau_2$ se e somente se $S(R_1) = S(R_2)$.*

Demonstração. Suponha que $S\tau_1 = S\tau_2$. Pelo lema 6.53, temos:

$$\begin{aligned} S(R_1) &= S([\![\tau_1]\!]) \\ &= [\![S\tau_1]\!] \\ &= [\![S\tau_2]\!] \\ &= S([\![\tau_2]\!]) \\ &= S(R_2) \end{aligned}$$

Assumindo que $S(R_1) = S(R_2)$, temos pelo lema 6.54 que:

$$\begin{aligned} S\tau_1 &= S([\![R_1]\!]^{-1}) \\ &= [\![S(R_1)]\!]^{-1} \\ &= [\![S(R_2)]\!]^{-1} \\ &= S([\![R_2]\!]^{-1}) \\ &= S\tau_2 \end{aligned}$$

□

Lema 6.56. *A aplicação de substituição é idempotente com respeito à função* fnames. *Formalmente, para todo S e R:*

$$\text{fnames}(S(R)) = \text{fnames}(R)$$

Demonstração.

$$\begin{aligned} \text{fnames}(S(R)) &= \{f \mid (f, \tau) \in S(R), \text{ para algum } \tau\} \\ &= \{f \mid (f, \tau) \in \{(f', S\tau') \mid (f', \tau') \in R\}, \text{ para algum } \tau\} \\ &= \{f \mid (f, \tau') \in R, \text{ para algum } \tau\} \\ &= \text{fnames}(R) \end{aligned}$$

□

Lema 6.57. cortypes(S(R), S(R′)) $\subseteq \Delta\mathbf{T}$ *se e somente se existe uma substituição S' tal que $S = S' \circ \text{mgu}(\text{cortypes}(R, R'))$.*

Demonstração. Seja cortypes(R, R′) $= \{(\tau_1, \tau_1'), \ldots, (\tau_n, \tau_n')\}$. Temos então:

$$\text{cortypes}(S(R), S(R')) = \{(S(\tau_1), S(\tau_1')), \ldots, (S(\tau_n), S(\tau_n'))\}$$

Como cortypes(S(R), S(R′)) $\subseteq \Delta\mathbf{T}$, então $S(\tau_i) = S(\tau_i')$ para $1 \leq i \leq n$. Portanto, S é um unificador dos pares de tipos de cortypes(R, R') e, pelas propriedades dos unificadores mais gerais (mgu), deve existir um unificador mais geral de cortypes(R, R') e uma substituição S' tais que $S = S' \circ \text{mgu}(\text{cortypes}(R, R'))$.

Por outro lado, se $S = S' \circ \text{mgu}(\text{cortypes}(R, R'))$ par algum S', temos:

$$\begin{aligned} &\text{cortypes}(S(R), S(R')) = \\ &\{(S(\tau_1), S(\tau_1')), \ldots, (S(\tau_n), S(\tau_n')\} = \\ &\{(S(\tau_1), S(\tau_1)), \ldots, (S(\tau_n), S(\tau_n)\} \subseteq \Delta\mathbf{T} \end{aligned}$$

□

Definição 6.58. A aplicação de uma substituição S a um termo de conjunto de campos t é uma operação parcial, que é definida se e somente se $S\alpha$ é uma variável de tipo ou um tipo registro, para todo $r_\alpha \in \mathrm{fv}(t)$. A aplicação de uma substituição a termos de conjuntos de campos é definida como:

$$St = \begin{cases} S(R) & \text{se } t = R \\ St_1 \oplus St_2 & \text{se } t = t_1 \oplus t_2 \text{ e se } St_1 \text{ e } St_2 \text{ são definidos} \\ St_1 \ominus St_2 & \text{se } t = t_1 \ominus t_2 \text{e se } St_1 \text{ e } St_2 \text{ são definidos} \\ [\![S\alpha]\!] & \text{se } t = r_\alpha \text{ e se } S\alpha \text{ é um tipo registro} \\ r_\beta & \text{se } t = r_\alpha \text{ e se } S\alpha = \beta, \text{ onde } \beta \text{ é uma variável de tipo} \\ \text{indefinido} & \text{se } t = r_\alpha \text{ e se } S\alpha \text{ não é nem um tipo registro ou uma variável de tipo} \end{cases}$$

Uma substituição S é considerada *ground* com respeito a um termo t se e somente se St é um termo *ground*. Similarmente, uma substituição S é dita válida com respeito a um termo t se e somente se St é um termo válido.

Lema 6.59. *Se t é um termo de conjunto de campos e S é uma substituição tal que St seja definido, então* $\mathrm{fv}(St) = \{r_\beta \mid r_\alpha \in \mathrm{fv}(t) \text{ e } S\alpha = \beta\}$.

Demonstração. Por indução no tamanho dos termos, suponha que o lema seja verdade para todo termo t' tal que $\mathrm{size}(t') < \mathrm{size}(t)$. Teríamos então, pela definição de fv, que:

- O lema é verdadeiro para $t = R$:

Trivialmente, pois $\mathrm{fv}(R) = \emptyset = \mathrm{fv}(S(R))$.

- O lema é verdadeiro para $t = r_\alpha$, para qualquer α:

Pela definição de fv, $\mathrm{fv}(t) = \mathrm{fv}(r_\alpha) = \{r_\alpha\}$. Suponha que $S\alpha = \beta$, para algum β. Teríamos então que $\mathrm{fv}(St) = \mathrm{fv}(Sr_\alpha) = \mathrm{fv}(r_\beta) = \{r_\beta\}$.

- O lema é verdadeiro para $t = t_1 \oplus t_2$ ou $t = t_1 \ominus t_2$:

Pela hipótese de indução, temos que o lema é verdadeiro para t_1 e t_2.

Suponha que $r_\alpha \in \mathrm{fv}(t)$ e que $S\alpha = \beta$, para algum β.

Pela definição de fv, $r_\alpha \in \mathrm{fv}(t_1)$ ou $r_\alpha \in \mathrm{fv}(t_2)$. Suponha, sem perda de generalidade, que $r_\alpha \in \mathrm{tv}(t_1)$. Então, pela hipótese de indução, temos que $r_\beta \in \mathrm{tv}(St_1)$ e, pela definição de fv, que $r_\beta \in \mathrm{tv}(t)$.

Por outro lado, suponha que $r_\beta \in \mathrm{fv}(St)$, para algum β. Então, pela definição de St e de fv, temos que $r_\beta \in St_1$ ou $r_\beta \in St_2$. Novamente, sem perda de generalidade, suponhamos que $r_\beta \in St_1$. Pela hipótese de indução, temos que deve haver um α tal que $r_\alpha \in \mathrm{fv}(t_1)$. Portanto, pela definição de fv, temos que $r_\alpha \in t$.

□

Lema 6.60. *Se t é um termo de conjunto de campo e S é uma substituição tal que St é* ground*, então, para toda variável de tipo α tal que $r_\alpha \in \mathrm{fv}(t)$, temos que $S\alpha$ é um tipo registro.*

Demonstração. Suponha que St seja *ground* e que tenha uma variável α tal que $r_\alpha \in \mathrm{fv}(t)$ e que $S\alpha$ não seja um tipo registro. Pela definição 6.58, $S\alpha = \beta$ para alguma variável β. Neste caso, pelo lema 6.59, temos que $r_\beta \in \mathrm{fv}(St)$ e, portanto, St não é um termo *ground*, o que contradiz a hipótese. □

Lema 6.61. *Se t é um termo de conjunto de campo e se S é uma substituição que, para toda variável α tal que $r_\alpha \in \mathrm{fv}(t)$, $S\alpha$ é um tipo registro, então S é* ground *com respeito a t.*

Demonstração. Como não existe $r_\alpha \in \mathrm{fv}(t)$ tal que $S\alpha = \beta$ para alguma variável β, então, pelo lema 6.59, temos que $\mathrm{fv}(St) = \emptyset$ e, portanto, St é um termo *ground*. □

Lema 6.62. *Se t é um termo de conjunto de campo e S é uma substituição, então St é* ground *se e somente se, para toda variável α tal que $r_\alpha \in \mathrm{fv}(t)$, $S\alpha$ é um tipo registro.*

Demonstração. Consequência direta dos lemas 6.60 e 6.61. □

6.5 Equações de conjuntos de campos

Definição 6.63. Uma equação de conjuntos de campos é um par de termos t_1 e t_2. Como o usual, escrevemos a equação como $t_1 \equiv t_2$. Os termos t_1 e t_2 são chamados, respectivamente, de termos esquerdo e termo direito da equação.

Uma equação é dita *ground* se os seus termos esquerdo e direito são termos *ground*. De semelhante modo, uma equação é dita *válida* se seus termos são termos válidos.

Definição 6.64. Se $e = t_1 \equiv t_2$ é uma equação de conjuntos de campos, e S é uma substituição, então Se é definido se e somente se St_1 e St_2 são definidos, em cujo caso temos:

$$Se = St_1 \equiv St_2$$

Definição 6.65. Dizemos que uma substituição S satisfaz uma equação $t_1 \equiv t_2$ se S é válida com respeito a t_1 e t_2, e se $\mathrm{val}(St_1) = \mathrm{val}(St_2)$, onde $val(t_1)$ e $\mathrm{val}(t_2)$ são os conjuntos de campos correspondentes a t_1 e t_2, respectivamente.

Lema 6.66. *Se S satisfaz a equação $r_\alpha \equiv t$, então $S\alpha$ tem de ser um tipo registro.*

Demonstração. A definição de satisfazibilidade de uma equação exige que $S(r_\alpha)$ seja um termo válido. Pela definição 6.58, sabemos que para que $S(r_\alpha)$ seja definido, $S\alpha$ precisa ser ou uma variável de tipo ou um tipo registro. Mas se $S\alpha$ é uma variável de tipo β, então $S(r_\alpha) = r_\beta$, o que não é um termo válido. Por outro lado, se $S\alpha$ é um tipo registro, então $S(r_\alpha) = [\![S\alpha]\!]$, que é um conjunto de campos e, portanto, um termo válido. □

Lema 6.67. *S satisfaz a equação $r_\alpha \equiv R$ se e somente se $S\alpha = S(\llbracket R \rrbracket^{-1})$.*

Demonstração. Pelo lema 6.66, sabemos que $S\alpha$ precisa ser um tipo registro. Temos então que $\llbracket S\alpha \rrbracket = S(R)$ e que:

$$\begin{aligned} S\alpha &= \left[\!\left[\llbracket S\alpha \rrbracket \right]\!\right]^{-1} \\ &= \llbracket S(R) \rrbracket^{-1} \\ &= S(\llbracket R \rrbracket^{-1}) \end{aligned}$$

Por outro lado, suponha que $S\alpha = S(\llbracket R \rrbracket^{-1})$. Como $S(\llbracket R \rrbracket^{-1})$ é um tipo registro, então $S(R_\alpha) \equiv S(R)$ é uma equação válida e, portanto:

$$\begin{aligned} S(R_\alpha) &= \llbracket S\alpha \rrbracket \\ &= \left[\!\left[S(\llbracket R \rrbracket^{-1}) \right]\!\right] \\ &= S(\left[\!\left[\llbracket R \rrbracket^{-1} \right]\!\right] \\ &= S(R) \end{aligned}$$

□

Teorema 6.68. *S satisfaz a equação $r_\alpha \equiv (r_\beta \ominus R_1) \oplus R_2$ se e somente se as seguintes condições forem verdadeiras:*

Condição 6.69. $S\alpha$ e $S\beta$ devem ser tipos registro.

Condição 6.70. $S(R_1) \subseteq \llbracket S\beta \rrbracket$

Condição 6.71. $(\text{fnames}(R_2) - \text{fnames}(R_1)) \cap \text{fnames}(\llbracket S\beta \rrbracket) = \emptyset$.

Condição 6.72. $S\alpha = \left[\!\left[(\llbracket S\beta \rrbracket / \text{fnames}(R_1)) \cup S(R_2) \right]\!\right]^{-1}$.

Condição 6.73. Se $\alpha = \beta$ então $S(R_1) = S(R_2)$.

Demonstração. Para provar a implicação direta, suponha que S satisfaz $r_\alpha \equiv (r_\beta \ominus R_1) \oplus R_2$. Isto significa que $S(r_\alpha)$ e $S((r_\beta \ominus R_1) \oplus R_2)$ são termos válidos e, portanto, termos *ground*.

Pelo lema 6.60, para que $S(r_\alpha)$ e $S((r_\beta \ominus R_1) \oplus R_2)$ sejam termos *ground*, $S\alpha$ e $S\beta$ precisam ser tipos registro, como exigido pela condição 6.69.

Pela definição 6.58, temos:

$$\begin{aligned} S(r_\alpha) &= \llbracket S\alpha \rrbracket \\ S((r_\beta \ominus R_1) \oplus R_2) &= (\llbracket S\beta \rrbracket \ominus S(R_1)) \oplus S(R_2) \end{aligned}$$

Para que $(\llbracket S\beta \rrbracket \ominus S(R_1)) \oplus S(R_2)$ seja um termo válido, as expressões $\llbracket S\beta \rrbracket \ominus S(R_1)$ e $(\llbracket S\beta \rrbracket \ominus S(R_1)) \oplus S(R_2)$ precisam ter um valor definido. Pela definição 6.18, $\llbracket S\beta \rrbracket \ominus S(R_1)$ é definido se e somente se $S(R_1) \subseteq \llbracket S\beta \rrbracket$, como exigido pela condição 6.70.

Se supormos que $[\![S\beta]\!] \ominus S(R_1)$ é definido, então teremos, pela definição 6.18, que $([\![S\beta]\!] \ominus S(R_1)) \oplus S(R_2)$ é definido se e somente se $\text{fnames}([\![S\beta]\!] \ominus S(R_1)) \cap \text{fnames}(S(R_2)) = \emptyset$. Pelos lemas 6.56 e 6.24, temos:

$$\begin{aligned}
\emptyset &= \text{fnames}([\![S\beta]\!] \ominus S(R_1)) \cap \text{fnames}(S(R_2)) \\
&= (\text{fnames}([\![S\beta]\!]) - \text{fnames}(S(R_1))) \cap \text{fnames}(S(R_2)) \\
&= (\text{fnames}([\![S\beta]\!]) - \text{fnames}(R_1)) \cap \text{fnames}(R_2) \\
&= \text{fnames}([\![S\beta]\!]) \cap (\text{fnames}(R_2) - \text{fnames}(R_1)) \\
&= (\text{fnames}(R_2) - \text{fnames}(R_1)) \cap \text{fnames}([\![S\beta]\!])
\end{aligned}$$

conforme exigido pela condição 6.71.

Para que a equação seja satisfeita, é necessário que $[\![S\alpha]\!] = ([\![S\beta]\!] \ominus R_1) \oplus R_2$. Pela definição 6.17 e pelos lemas 6.20 e 6.56, temos:

$$\begin{aligned}
S\alpha &= [\![\, [\![S\alpha]\!] \,]\!]^{-1} \\
&= [\![([\![S\beta]\!] \ominus S(R_1)) \oplus S(R_2)]\!]^{-1} \\
&= [\![([\![S\beta]\!]/\text{fnames}(S(R_1))) \oplus S(R_2)]\!]^{-1} \\
&\quad [\![([\![S\beta]\!]/\text{fnames}(R_1)) \oplus S(R_2)]\!]^{-1} \\
&\quad [\![([\![S\beta]\!]/\text{fnames}(R_1)) \cup S(R_2)]\!]^{-1}
\end{aligned}$$

como exigido pela condição 6.72.

No caso particular em que $\alpha = \beta$, o seguinte precisa ser verdade para que a equação possa ser satisfeita por S:

$$[\![S\alpha]\!] = ([\![S\alpha]\!] \ominus S(R_1)) \oplus S(R_2)$$

Pelas definições 6.17 e 6.18, sabemos que a seguinte equação tem que ser satisfeita:

$$[\![S\alpha]\!] = ([\![S\alpha]\!] - S(R_1)) \cup S(R_2)$$

Pela condição 6.70, sabemos que $S(R_1) \subseteq [\![S\alpha]\!]$. Podemos também deduzir da equação acima que $S(R_2) \subseteq [\![S\alpha]\!]$.

Temos então:

$$\begin{aligned}
[\![S\alpha]\!] &= ([\![S\alpha]\!] - S(R_1)) \cup S(R_2) \\
[\![S\alpha]\!] \cap S(R_1) &= (([\![S\alpha]\!] - S(R_1)) \cup S(R_2)) \cap S(R_1) \\
S(R_1) &= (([\![S\alpha]\!] - S(R_1)) \cap S(R_1)) \cup (S(R_1) \cap S(R_2)) \\
S(R_1) &= S(R_1) \cap S(R_2) \\
S(R_1) &\subseteq S(R_2)
\end{aligned}$$

Pela definição 6.17 e pelo lema 6.11, podemos deduzir que $([\![S\alpha]\!] - S(R_1)) \cap S(R_2) = \emptyset$. Usamos esta propriedade para isolar $S(R_1)$:

$$\begin{aligned}
[\![S\alpha]\!] &= ([\![S\alpha]\!] - S(R_1)) \cup S(R_2) \\
[\![S\alpha]\!] - S(R_2) &= [\![S\alpha]\!] - S(R_1) \\
([\![S\alpha]\!] - S(R_2)) \cup S(R_1) &= ([\![S\alpha]\!] - S(R_1)) \cup S(R_1) \\
([\![S\alpha]\!] - S(R_2)) \cup S(R_1) &= [\![S\alpha]\!]
\end{aligned}$$

Agora, usando um argumento simétrico ao usado para provar que $S(R_1) \subseteq S(R_2)$, podemos deduzir que $S(R_2) \subseteq S(R_1)$ e, portanto, que $S(R_1) = S(R_2)$, como exigido pela condição 6.73.

Para a implicação reversa, suponha que a substituição S e a equação $r_\alpha \equiv (r_\beta \ominus R_1) \oplus R_2$ satisfaçam as condições expressas acima. Teríamos então que: □

- $S\alpha$ é um tipo registro (pela condição 6.69) e, portanto, $S(r_\alpha)$ é definido e é igual a $[\![S\alpha]\!]$;
- $S\beta$ é um tipo registro (pela condição 6.69) e, portanto, $S(r_\beta)$ é definido e é igual a $[\![S\beta]\!]$;
- $S(R_1) \subseteq [\![S\beta]\!]$ (pela condição 6.70) e, portanto, $S(r_\beta \ominus R_1) = [\![S\beta]\!] \ominus S(R_1)$ é definido e é igual a $[\![S\beta]\!]/\mathrm{fnames}(S(R_1)) = [\![S\beta]\!]/\mathrm{fnames}(R_1)$;
- $(\mathrm{fnames}(R_2) - \mathrm{fnames}(R_1)) \cap \mathrm{fnames}([\![S\beta]\!]) = \emptyset$ (pela condição 6.71), o que nos permite deduzir que:

$$\begin{aligned}
\emptyset &= (\mathrm{fnames}(R_2) - \mathrm{fnames}(R_1)) \cap \mathrm{fnames}([\![S\beta]\!]) \\
&= (\mathrm{fnames}(S(R_2)) - \mathrm{fnames}(S(R_1))) \cap \mathrm{fnames}([\![S\beta]\!]) \\
&= \mathrm{fnames}(S(R_2)) \cap (\mathrm{fnames}([\![S\beta]\!]) - \mathrm{fnames}(S(R_1))) \\
&= (\mathrm{fnames}([\![S\beta]\!]) - \mathrm{fnames}(S(R_1))) \cap \mathrm{fnames}(S(R_2)) \\
&= \mathrm{fnames}([\![S\beta]\!] \ominus S(R_1)) \cap \mathrm{fnames}(S(R_2))
\end{aligned}$$

  Portanto, $([\![\mathrm{S}\beta]\!] \ominus \mathrm{S}(\mathrm{R}_1)) \oplus \mathrm{S}(\mathrm{R}_2)$ é definido e é igual a $([\![S\beta]\!]/\mathrm{fnames}(R_1)) \cup S(R_2)$;

- Finalmente, pela condição 6.72, temos:

$$S\alpha = \left[\!\left[\, ([\![S\beta]\!]/\mathrm{fnames}(R_1)) \cup S(R_2) \, \right]\!\right]^{-1}$$

  e

$$\begin{aligned}
S(r_\alpha) &= [\![S\alpha]\!] \\
&= ([\![S\beta]\!]/\mathrm{fnames}(R_1)) \cup S(R_2) \\
&= ([\![\mathrm{S}\beta]\!] \ominus \mathrm{S}(\mathrm{R}_1)) \oplus \mathrm{S}(\mathrm{R}_2) \\
&= (S(r_\beta) \ominus S(R_1)) \oplus S(R_2) \\
&= S((r_\beta \ominus R_1) \oplus R_2)
\end{aligned}$$

  e, portanto, S satisfaz a equação.

6.6 Sistemas de equações de conjuntos de campos

Definição 6.74. Um sistema de equações de conjuntos de campos é um conjunto finito E de equações de conjuntos de campos.

Definição 6.75. Uma substituição S satisfaz um sistema de equações de conjuntos de campos E se e somente se S satisfaz cada uma das equações de E. Chamamos de $\text{solset}(e)$ o conjunto de substituições que satisfazem a equação e e de $\text{solset}(E)$ o conjunto de substituições que satisfazem E. Temos o seguinte relacionamento entre as duas funções:

$$\text{solset}(E) = \bigcap_{e \in E} \text{solset}(e)$$

Em particular, $\text{solset}(\emptyset) = \mathbf{S}$, onde $\mathbf{S}$ é o conjunto de substituições.

Lema 6.76. $\text{solset}(E_1 \cup E_2) = \text{solset}(E_1) \cap \text{solset}(E_2)$.

Demonstração.

$$\begin{aligned} \text{solset}(E_1 \cup E_2) &= \bigcap_{e \in E_1 \cup E_2} \text{solset}(e) \\ &= \left(\bigcap_{e \in E_1} \text{solset}(e) \right) \cap \left(\bigcap_{e \in E_2} \text{solset}(e) \right) \\ &= \text{solset}(E_1) \cap \text{solset}(E_2) \end{aligned}$$

□

Lema 6.77. $\text{solset}(E)$ *é anti-monotônico com respeito a* $\subseteq$*: para todo* E_1 *e* E_2*, se* $E_1 \subseteq E_2$*, então* $\text{solset}(E_2) \subseteq \text{solset}(E_1)$.

Demonstração. Suponha que $E_1 \subseteq E_2$. Se $E_1 = E_2$ então $\text{solset}(E_1) = \text{solset}(E_2)$ e o lema é trivialmente verdadeiro.

Se $E_1 \subseteq E_2$então $E_2 = E_1 \cup E_2$. Neste caso, temos:

$$\begin{aligned} \text{solset}(E_2) &= \text{solset}(E_1 \cup E_2) \\ &= \text{solset}(E_1) \cap \text{solset}(E_2) \\ &\subseteq \text{solset}(E_1) \end{aligned}$$

□

Lema 6.78. *Se* $\text{solset}(E_1) = \text{solset}(E_1')$*, então* $\text{solset}(E_1 \cup E_2) = \text{solset}(E_1' \cup E_2)$.

Demonstração.

$$\begin{aligned} \text{solset}(E_1 \cup E_2) &= \text{solset}(E_1) \cap \text{solset}(E_2) \\ &= \text{solset}(E_1') \cap \text{solset}(E_2) \\ &= \text{solset}(E_1' \cup E_2) \end{aligned}$$

□

6.7 A forma $\preceq$-normal

Definição 6.79. Dada uma ordem total $\preceq$ sobre variáveis de tipo, um sistema de equações E está na forma $\preceq$-*normal* se e somente se as seguintes condições forem satisfeitas:

Condição 6.80. Toda equação de E deve ter a forma:

$$r_\alpha \equiv \left(r_\beta \ominus R^-\right) \oplus R^+$$

Dada uma enumeração arbitrária das equações de E, usaremos a seguinte notação para representar a i-ésima equação de E, onde $1 \leq i \leq n$ e $n = |E|$:

$$r_{\alpha_i} \equiv \left(r_{\beta_i} \ominus R_i^-\right) \oplus R_i^+$$

.

Condição 6.81. $\beta_i \preceq \alpha_i$

Condição 6.82. $R_i^- \cup R_i^+ \neq \emptyset$.

Condição 6.83. $\alpha_i = \beta_i$ se e somente se $R_i^- = R_i^+$.

Condição 6.84. Se $\alpha_i = \alpha_j$ então $i = j$.

Condição 6.85. *Se* $\beta_i = \beta_j$, $R_i^- = R_j^-$ e $R_i^+ = R_j^+$, então $i = j$.

Condição 6.86. Se $\alpha_i = \beta_j$ então $i = j$.

Condição 6.87. Se $\beta_i = \beta_j$ então $\text{cortypes}(R_i^-, R_j^-) \subseteq \mathbf{\Delta T}$.

Condição 6.88. Se $\beta_i = \beta_j$ então $\text{fnames}(R_i^-) \cap \text{fnames}(R_j^+) \subseteq \text{fnames}(R_j^-)$.

As variáveis de tipo que ocorrem em sistema de equações $\preceq$-normal são classificadas de acordo com a posição em que ocorrem em equações de E:

- α é uma variável direita de E se ocorre como uma variável de registro (r_α) no termo direito de uma equação de E. Em outras palavras deve existir em E uma equação com a forma:
$$r_\beta \equiv (r_\alpha \ominus R^-) \oplus R^+$$
- α é uma variável esquerda de E se não for uma variável direita e se ocorrer no termo esquerdo de uma equação de E como uma variável de registro. Em outras palavras, deve haver uma equação em E com a seguinte forma:
$$r_\alpha \equiv (r_\beta \ominus R^-) \oplus R^+$$

A condição 6.84 garante que se α ocorre do lado esquerdo de uma equação de E, então ela só ocorre no lado esquerdo de uma única equação de E. Portanto, existe apenas uma variável direita correspondente a cada variável esquerda α de E, denotada por $\mathrm{rvar}(\alpha)$.

De modo semelhante, usamos a notação $\mathrm{posR}(\alpha, E)$ e $\mathrm{negF}(\alpha, E)$ para denotar, respectivamente, o conjunto de campos R^+ e o conjunto de nomes de campos $\mathrm{fnames}(R^-)$ da equação onde α aparece do lado esquerdo.

- α é uma variável de campo de E se ocorre em E, porém não é nem uma variável esquerda nem uma variável direita.

Definição 6.89. Chamamos de $\mathrm{RV}(E)$, $\mathrm{LV}(E)$ e $\mathrm{FV}(E)$, respectivamente, os conjuntos de variáveis direitas, esquerdas e de campo de E.

Definição 6.90. As seguintes funções são definidas para sistemas de equação $\preceq$-normais, e para variáveis direitas destes sistemas:

$$\begin{aligned}
\mathrm{fields}(\alpha, E) &= \bigcup\{R^- \mid r_\beta \equiv (r_\alpha \ominus R^-) \oplus R^+ \in E\} \\
\mathrm{negfnames}(\alpha, E) &= \bigcup\{\mathrm{fnames}(R^+) - \mathrm{fnames}(R^-) \mid r_\beta \equiv (r_\alpha \ominus R^-) \oplus R^+ \in E\} \\
\mathrm{posfnames}(\alpha, E) &= \bigcup\{\mathrm{fnames}(R^-) \mid r_\beta \equiv (r_\alpha \ominus R^-) \oplus R^+ \in E\} \\
\mathrm{allfnames}(\alpha, E) &= \mathrm{negfnames}(\alpha, E) \cup \mathrm{posfnames}(\alpha, E)
\end{aligned}$$

Para cada E e α, onde α é uma variável direita de E, os valores destas funções correspondem, respectivamente a: campos que E exige que α deve ter ($\mathrm{fields}(\alpha, E)$); nomes dos campos que E exige que α não pode ter; nomes dos campos que E exige que α deva ter ($\mathrm{posfnames}(\alpha, E)$); e os nomes de campos que E exige queα deva ter ou não ter ($\mathrm{allfnames}(\alpha, E)$).

Definição 6.91. sPara cada variável de registro r_α de E (ou uma variável esquerda ou direita de E) definimos a função $\mathrm{subvars}(\alpha, E)$ como:

$$\begin{aligned}
\mathrm{subvars}(\alpha, E) &= \mathrm{subvars}(\alpha, E, \emptyset) \\
\mathrm{subvars}(\alpha, E, F) &= \left\{ \alpha' \mid \begin{array}{l} r_{\alpha_1} \equiv (r_{\alpha_2} \ominus R^-) \oplus R^+ \text{ e} \\ \quad ((\alpha = \alpha_1 \text{ e } \alpha' \in \mathrm{subvars}_L(\alpha, E, R^+, R^-, F)) \text{ ou} \\ \quad (\alpha = \alpha_2 \text{ e } \alpha' \in \mathrm{subvars}_R(\alpha, E, R^-, R^+, F))) \end{array} \right\} \\
\mathrm{subvars}(\alpha, E, R_1, R_2, F) &= \left\{ \alpha' \mid \begin{array}{l} \alpha'' \in \mathrm{tv}(R_1) \text{ e} \\ \quad (\alpha' = \alpha'' \text{ or } \alpha' \in \mathrm{subvars}(\alpha, E, F \cup \mathrm{fnames}(R_2))) \end{array} \right\}
\end{aligned}$$

Esta função denota, para cada variável de registro de E, o conjunto de variáveis de tipo que E exige que ocorram em instâncias de α (subvariáveis de α em E). Esta função é recursiva, pois E pode exigir que uma variável ocorra em α de maneira indireta, como quando E exige que α tenha o campo (f, β_1) e também exige que β_1 tenha o campo (f, β_2).

Definição 6.92. Para cada variável direita α de E, existe um conjunto de nomes de campos chamado conjunto de nomes livres de α em E. Este conjunto consiste nos nomes de campos que não restringidos por E em relação a α. Nomes deste conjunto não têm sua ocorrência em α exigida nem proibida por E. Formalmente:

$$\text{freefnames}(\alpha,\, E) = \mathbf{F} - \text{allfnames}(\alpha,\, E)$$

onde $\mathbf{F}$ é o conjunto de todos os nomes de campos.

Similarmente, definimos o conjunto de conjuntos de campos livres de α em E, que consiste nos conjuntos de campos que não possuem campos que são de alguma forma relacionados a α por E, e que não possuem subvariáveis de α em E:

$$\text{freefvalues}(\alpha,\, E) = \left\{ R \mid \begin{array}{l} F \subseteq \text{freefnames}(\alpha,\, E),\ F \text{ é finito,} \\ h : F \to \mathbf{T}/\text{subvars}(\alpha,\, E) \text{ e} \\ R = \{(f,\, h(f)) \mid f \in \mathrm{F}\} \end{array} \right\}$$

onde $\mathbf{T}/\text{subvars}(\alpha,\, E)$ é o conjunto de tipos simples τ que não incluem subvariáveis de α em E:

$$\mathbf{T}/\text{subvars}(\alpha,\, E) = \{\tau \mid \tau \in \mathbf{T} \text{ e } \text{tv}(\tau) \cap \text{subvars}(\alpha,\, E) = \emptyset\}$$

Lema 6.93. *Para cada sistema de equação $\leq$-normal E e para cada variável $\alpha \in \text{RV}(E)$, temos que $\emptyset \in \text{freefvalues}(\alpha,\, E)$.*

Demonstração. Basta fazer $F = \emptyset$ na definição acima. □

Lema 6.94. *Se E é um sistema de equação $\preceq$-normal, então para α, $\text{fields}(\alpha,\, E)$ é um conjunto de campos.*

Demonstração. Pela definição 6.90, $\text{fields}(\alpha,\, E)$ é uma união de conjuntos de campos. Se não possuir campos distintos com o mesmo nome, $\text{fields}(\alpha,\, E)$ também é um conjunto de campos, como definido em 6.1.

Suponha que $(f,\, \tau) \in \text{fields}(\alpha,\, E)$, que $(f,\, \tau') \in \text{fields}(\alpha,\, E)$ e que $\tau \neq \tau'$.

Pela definição 6.90, temos que devem haver β_1, β_2, R_1^-, R_2^-, R_1^+ e R_2^+ tais que:

$$\begin{array}{rcl} r_{\beta_1} & \equiv & (r_\alpha \ominus R_1^-) \oplus R_2^+ \\ (f,\, \tau) & \in & R_1^- \\ r_{\beta_2} & = & (r_\alpha \ominus R_1^-) \oplus R_2^+ \\ (f,\, \tau') & \in & R_2^- \end{array}$$

Mas a condição 6.87 nos garante que $\text{cortypes}(R_1^-,\, R_2^-) \subseteq \Delta\mathbf{T}$, o que significa que se R_1^- e R_2^- tiverem campos com o mesmo nome, estes devem ter o mesmo tipo, ou seja, $\tau = \tau'$, o que contradiz a hipótese. □

Definição 6.95. Para cada sistema de equações $\leq$-normal E, seja $\mathrm{normsol}(E)$ o conjunto de substituições definido como:

$$\begin{aligned}
\mathrm{normsol}(E) &= \left\{ S \circ S^h \mid \begin{array}{l} h : \mathrm{RV}(E) \to \mathbf{R} \text{ e} \\ \quad \text{para todo } \alpha \in \mathrm{RV}(E),\ h(\alpha) \in \mathrm{freefvalues}(\alpha, E) \end{array} \right\} \\
S^h &= \mathrm{mgu} \left(\begin{array}{l} \{(\alpha, [\![R^{\alpha,h}]\!]^{-1}) \mid \alpha \in \mathrm{RV}(E)\} \cup \\ \{(\alpha, [\![(R^{\mathrm{rvar}(\alpha,E),h}/\mathrm{negF}(\alpha, E)) \cup \mathrm{posR}(\alpha, E)]\!]^{-1}) \mid \alpha \in \mathrm{LV}(E) \end{array} \right) \\
R^{\alpha,h} &= \mathrm{fields}(\alpha, E) \cup h(\alpha)
\end{aligned}$$

Teorema 6.96. $\mathrm{normsol}(E)$ *é não-vazio se e somente se, para toda variável de registro* r_α *de* E, α *não é auto-recursiva, ou seja,* $\alpha \notin \mathrm{subvars}(\alpha, E)$.

Demonstração. Informalmente, se $\mathrm{normsol}(E)$ é não-vazio, então deve existir uma substituição S^h que satisfaça a definição acima. Para que essa substituição possa existir, deve existir também o unificador mais geral (*mgu*) do conjunto de tipos:

$$\begin{array}{l} \{(\alpha, [\![R^{\alpha}]\!]^{-1}) \mid \alpha \in \mathrm{RV}(E)\} \cup \\ \{(\alpha, [\![(R^{\mathrm{rvar}(\alpha,E)}/\mathrm{negF}(\alpha, E)) \cup \mathrm{posR}(\alpha, E)]\!]^{-1}) \mid \alpha \in \mathrm{LV}(E)\} \end{array}$$

Este unificador só existe se nenhuma variável em $\mathrm{RV}(E)$ ou em $\mathrm{LV}(E)$ tenha de ser unificada com um tipo que contém, direta, ou indiretamente, esta mesma variável. Se uma variável auto-recursiva estiver presente em E, este unificador teria que substituir esta variável por um termo de tipo infinito. □

Lema 6.97. $\mathrm{normsol}(E) \subseteq \mathrm{solset}(E)$.

Demonstração. Suponha que $r_{\alpha_L} \equiv (r_{\alpha_R} \ominus R^-) \oplus R^+$ é uma equação de E, e que $S \in \mathrm{normsol}(E)$. Pela definição 6.95, devem haver substituições S_1 e S_2 e uma função $h : \mathrm{RV}(E) \to \mathbf{R}$ tais que:

$$\begin{aligned}
S &= S_2 \circ S_1 \\
h(\alpha) &\in \mathrm{freefvalues}(\alpha, E), \text{ para todo } \alpha \in \mathrm{RV}(E) \\
S_1 &= \mathrm{mgu} \left(\begin{array}{l} \{(\alpha, [\![R^{\alpha,h}]\!]^{-1}) \mid \alpha \in \mathrm{RV}(E)\} \cup \\ \{(\alpha, [\![(R^{\mathrm{rvar}(\alpha,E),h}/\mathrm{negF}(\alpha, E)) \cup \mathrm{posR}(\alpha, E)]\!]^{-1}) \mid \alpha \in \mathrm{LV}(E) \end{array} \right) \\
R^{\alpha,h} &= \mathrm{fields}(\alpha, E) \cup h(\alpha)
\end{aligned}$$

Temos então que:

$$\begin{aligned}
[\![S\alpha_R]\!] &= [\![S_2(S_1\alpha_R)]\!] \\
&= [\![S_2(S_1[\![R^{\alpha_R,h}]\!]^{-1})]\!] \\
&= S_2([\![S_1[\![R^{\alpha_R,h}]\!]^{-1}]\!]) \\
&= S_2(S_1([\![\,[\![R^{\alpha_R,h}]\!]^{-1}\,]\!])) \\
&= S_2(S_1(R^{\alpha_R,h})) \\
&= \{(f,\, S_2(S_1\tau)) \mid (f,\, \tau) \in R^{\alpha_R,h}\} \\
&= S_2(S_1(\text{fields}(\alpha_R,\, E))) \cup S_2(S_1(h(\alpha_R)))
\end{aligned}$$

e também que:

$$\begin{aligned}
[\![S\alpha_L]\!] &= [\![S_2(S_1\alpha_L)]\!] \\
&= [\![S_2(S_1[\![(R^{\alpha_R,h}/\text{negF}(\alpha_L,\, E)) \cup \text{posR}(\alpha_L,\, E)]\!]^{-1})]\!] \\
&= S_2([\![S_1[\![(R^{\alpha_R,h}/\text{negF}(\alpha_L,\, E)) \cup \text{posR}(\alpha_L,\, E)]\!]^{-1}]\!]) \\
&= S_2(S_1([\![\,[\![(R^{\alpha_R,h}/\text{negF}(\alpha_L,\, E)) \cup \text{posR}(\alpha_L,\, E)]\!]^{-1}\,]\!])) \\
&= S_2(S_1((R^{\alpha_R,h}/\text{negF}(\alpha_L,\, E)) \cup \text{posR}(\alpha_L,\, E))) \\
&= S_2(S_1(R^{\alpha_R,h}/\text{negF}(\alpha_L,\, E))) \cup S_2(S_1(\text{posR}(\alpha_L,\, E))) \\
&= (S_2(S_1(\text{fields}(\alpha_R,\, E))) \cup S_2(S_1(h(\alpha_R))))/\text{negF}(\alpha_L,\, E)) \cup S_2(S_1(\text{posR}(\alpha_L,\, E))) \\
&= (S_2(S_1(\text{fields}(\alpha_R,\, E))) \cup S_2(S_1(h(\alpha_R))))/\text{fnames}(R^-)) \cup S_2(S_1(R^+))
\end{aligned}$$

Pela definição 6.92, sabemos que:

$$\begin{aligned}
\text{fnames}(h(\alpha_R)) &\subseteq \text{freefnames}(\alpha_R,\, E) \\
&\subseteq \mathbf{F} - \text{allfnames}(\alpha_R,\, E) \\
&\subseteq \mathbf{F} - \text{posfnames}(\alpha_R,\, E) - \text{negfnames}(\alpha_R,\, E) \\
&\subseteq \mathbf{F} - \text{fnames}(\text{fields}(\alpha_R,\, E))
\end{aligned}$$

e que:

$$\begin{aligned}
\text{fnames}(h(\alpha_R)) &\subseteq \mathbf{F} - \text{posfnames}(\alpha_R,\, E) - \text{negfnames}(\alpha_R,\, E) \\
&\subseteq \mathbf{F} - \text{fnames}(R^+)
\end{aligned}$$

Portanto, temos que $\text{fnames}(h(\alpha_R)) \cap \text{fnames}(\text{fields}(\alpha_R,\, E)) = \emptyset$ e que $\text{fnames}(h(\alpha_R)) \cap \text{fnames}(R^+) = \emptyset$.

Da condição 6.88 , podemos deduzir que:

$$\text{fnames}(R^+) \cap (\text{fnames}(\text{fields}(\alpha_R,\, E)) \cup \text{fnames}(h(\alpha_R))) = \emptyset$$

Levando em conta que, pela definição 6.90, $R^- \subseteq \text{fields}(\alpha_R,\, E)$, temos:

$$\begin{aligned}
S\alpha_L &= [\![S\alpha_L]\!] \\
&= (S_2(S_1(\text{fields}(\alpha_R,\, E))) \cup S_2(S_1(h(\alpha_R))))/\text{fnames}(R^-)) \cup S_2(S_1(R^+)) \\
&= (S_2(S_1(\text{fields}(\alpha_R,\, E))) \cup S_2(S_1(h(\alpha_R)))) \ominus S_2(S_1(R^-))) \cup S_2(S_1(R^+)) \\
&= (S_2(S_1(\text{fields}(\alpha_R,\, E))) \cup S_2(S_1(h(\alpha_R)))) \ominus S_2(S_1(R^-))) \oplus S_2(S_1(R^+)) \\
&= ([\![S\alpha_R]\!]^{-1} \ominus S(R^-)) \oplus S(R^+) \\
&= S((\alpha_R \ominus R^-) \oplus R^+)
\end{aligned}$$

Portanto, S é uma solução para a equação $\alpha_L \equiv (\alpha_R \ominus R^-) \oplus R^+$, e, por generalização, de todas as equações de E. □

Lema 6.98. $\text{solset}(E) \subseteq \text{normsol}(E)$.

Demonstração. Suponha que S seja uma solução de E. Isto significa que, para toda variável de registro α de E, $S\alpha$ é um tipo registro, e para toda equação $r_{\alpha_L} \equiv (r_{\alpha_R} \ominus R^-) \oplus R^+$, o seguinte é verdade:

$$[\![S\alpha_L]\!]^{-1} = ([\![S\alpha_R]\!]^{-1} \ominus S(R^-)) \oplus S(R^+)$$

Para que $[\![S\alpha_R]\!]^{-1} \ominus S(R^-)$ seja definido, é preciso que $S(R^-) \subseteq [\![S\alpha_R]\!]^{-1}$. Generalizando esta condição para todas as equações de E, temos, pela definição 6.90, que:

$$S(\text{fields}(\alpha_R,\, E)) \subseteq [\![S\alpha_R]\!]^{-1}$$

Portanto, deve haver um conjunto de campos R_{α_R} tal que $R_{\alpha_R} \cap S(\text{fields}(\alpha_R,\, E)) = \emptyset$ e tal que $R_{\alpha_R} \cup S(\text{fields}(\alpha_R,\, E)) = [\![S\alpha_R]\!]^{-1}$. Como $[\![S\alpha_R]\!]^{-1}$ é um conjunto de campos e, portanto, não pode ter dois campos com o mesmo nome, o seguinte deve ser verdade:

$$\begin{aligned}
&\text{fnames}(R_{\alpha_R}) \cap \text{fnames}(\text{fields}(\alpha_R,\, E)) = \emptyset \text{ e} \\
&R_{\alpha_R} \cup S(\text{fields}(\alpha_R,\, E)) = R \oplus S(\text{fields}(\alpha_R,\, E))
\end{aligned}$$

Para que $([\![S\alpha_R]\!]^{-1} \ominus S(R^-)) \oplus S(R^+)$ seja definido, o seguinte precisa ser verdade:

$$\text{fnames}(([\![S\alpha_R]\!]^{-1} \ominus S(R^-)) \cap \text{fnames}(S(R^+)) = \emptyset$$

Mas, pelas propriedades da aplicação de substituição e dos operadores $\oplus$ e $\ominus$, temos:

$$\begin{aligned}
\emptyset &= \text{fnames}(([\![S\alpha_R]\!]^{-1} \ominus S(R^-)) \cap \text{fnames}(S(R^+)) \\
&= (\text{fnames}([\![S\alpha_R]\!]) - \text{fnames}(R^-)) \cap \text{fnames}(R^+) \\
&= (\text{fnames}(R_{\alpha_R} \oplus S(\text{fields}(\alpha_R,\, E))) - \text{fnames}(R^-)) \cap \text{fnames}(R^+) \\
&= ((\text{fnames}(R_{\alpha_R}) \cup \text{fnames}(\text{fields}(\alpha_R,\, E))) - \text{fnames}(R^-)) \cap \text{fnames}(R^+)
\end{aligned}$$

Como $\mathrm{R}^- \subseteq \mathrm{fields}(\alpha_R,\, E)$, então $\mathrm{fnames}(R^-) \subseteq \mathrm{fnames}(\mathrm{fields}(\alpha_R,\, E))$. Como $\mathrm{fnames}(R_{\alpha_R}) \cap \mathrm{fnames}(\mathrm{fields}(\alpha_R,\, E)) = \emptyset$, então $\mathrm{fnames}(R^-) \cap \mathrm{fnames}(R_{\alpha_R}) = \emptyset$ e:

$$\begin{aligned}
\emptyset &= ((\mathrm{fnames}(R_{\alpha_R}) \cup \mathrm{fnames}(\mathrm{fields}(\alpha_R,\, E))) - \mathrm{fnames}(R^-)) \cap \mathrm{fnames}(R^+) \\
&= (\mathrm{fnames}(R_{\alpha_R}) \cup (\mathrm{fnames}(\mathrm{fields}(\alpha_R,\, E)) - \mathrm{fnames}(R^-))) \cap \mathrm{fnames}(R^+) \\
&= (\mathrm{fnames}(R_{\alpha_R}) \cap \mathrm{fnames}(R^+)) \cup ((\mathrm{fnames}(\mathrm{fields}(\alpha_R,\, E)) - \mathrm{fnames}(R^-)) \cap \mathrm{fnames}(R^+)) \\
&\supseteq \mathrm{fnames}(R_{\alpha_R}) \cap \mathrm{fnames}(R^+)
\end{aligned}$$

Portanto, $\mathrm{fnames}(R_{\alpha_R}) \cap \mathrm{fnames}(\mathrm{R}^+) \subseteq \emptyset$, e $\mathrm{fnames}(R_{\alpha_R}) \cap (\mathrm{fnames}(R^-) \cup \mathrm{fnames}(R^+)) = \emptyset$.

Generalizando este resultado para todas as equações, temos:

$$\mathrm{fnames}(R_{\alpha_R}) \cap \mathrm{allfnames}(\alpha_R,\, E) = \emptyset$$

Retornando à condição que garante que S é uma solução de cada equação em E, temos:

$$\begin{aligned}
[\![S\alpha_L]\!]^{-1} &= ([\![S\alpha_R]\!]^{-1} \ominus S(R^-)) \oplus S(R^+) \\
&= ([\![S\alpha_R]\!]^{-1}/\mathrm{fnames}(R^-)) \cup S(R^+) \\
&= ((R_{\alpha_R} \cup S(\mathrm{fields}(\alpha_R,\, E)))/\mathrm{fnames}(R^-)) \cup S(R^+)
\end{aligned}$$

Fazendo $h(\alpha_R) = R'_{\alpha_R}$, onde $S(R'_{\alpha_R}) = R_{\alpha_R}$, temos:

$$\begin{aligned}
[\![S\alpha_R]\!]^{-1} &= S(h(\alpha_R)) \cup S(\mathrm{fields}(\alpha_R,\, E)) \\
[\![S\alpha_L]\!]^{-1} &= ((S(h(\alpha_R)) \cup S(\mathrm{fields}(\alpha_R,\, E)))/\mathrm{fnames}(R^-)) \cup S(R^+)
\end{aligned}$$

Aplicando $[\![\,]\!]^{-1}$ a ambos os lados, obtemos:

$$\begin{aligned}
S\alpha_R &= [\![S(h(\alpha_R)) \cup S(\mathrm{fields}(\alpha_R,\, E))]\!]^{-1} \\
S\alpha_L &= [\![((S(h(\alpha_R)) \cup S(\mathrm{fields}(\alpha_R,\, E)))/\mathrm{fnames}(R^-)) \cup S(R^+)]\!]^{-1}
\end{aligned}$$

Mas $\alpha_R = \mathrm{rvar}(\alpha_L,\, E)$, $\mathrm{fnames}(R^-) = \mathrm{negF}(\alpha_L,\, E)$ e $R^+ = \mathrm{posR}(\alpha_L,\, E)$, o que nos permite reescrever as equações acima como:

$$\begin{aligned}
S\alpha_R &= [\![S(h(\alpha_R)) \cup S(\mathrm{fields}(\alpha_R,\, E))]\!]^{-1} \\
&= [\![S(h(\alpha_R) \cup S(\mathrm{fields}(\alpha_R,\, E)))]\!]^{-1} \\
&= [\![S(h(\alpha_R) \cup \mathrm{fields}(\alpha_R,\, E))]\!]^{-1} \\
&= S([\![h(\alpha_R) \cup \mathrm{fields}(\alpha_R,\, E)]\!]^{-1}) \\
&= S([\![R^{\alpha_R,\, h}]\!]^{-1}) \\
S\alpha_L &= [\![((S(h(\mathrm{rvar}(\alpha_L,\, E))) \cup S(\mathrm{fields}(\mathrm{rvar}(\alpha_L,\, E)))/\mathrm{negF}(\alpha_L,\, E)) \cup S(\mathrm{posR}(\alpha_L,\, E))]\!]^{-1} \\
&= [\![S((R^{\mathrm{rvar}(\alpha,\, E),h}/\mathrm{negF}(\alpha_L,\, E)) \cup \mathrm{posR}(\alpha_L,\, E))]\!]^{-1} \\
&= S([\![(R^{\mathrm{rvar}(\alpha,\, E),h}/\mathrm{negF}(\alpha_L,\, E)) \cup \mathrm{posR}(\alpha_L,\, E)]\!]^{-1})
\end{aligned}$$

onde $R^{\alpha,h} = \text{fields}(\alpha,\, E) \cup h(\alpha)$.

Portanto, S é um unificador do seguinte conjunto de pares de tipos:

$$\{(\alpha,\, [\![R^{\alpha,\,h}]\!]^{-1}) \mid \alpha \in \text{RV}(E)\} \cup$$
$$\{(\alpha,\, [\![(R^{\text{rvar}(\alpha,\,E),\,h}/\text{negF}(\alpha,\, E)) \cup \text{posR}(\alpha,\, E)]\!]^{-1}) \mid \alpha \in \text{LV}(E)\}$$

Portanto, pela definição do unificador mais geral de um conjunto de pares de tipos, devem haver substituições S_2 e S_1 tais que $S = S_2 \circ S_1$, onde:

$$S_1 = \text{mgu}\left(\begin{array}{l} \{(\alpha,\, [\![R^{\alpha,h}]\!]^{-1}) \mid \alpha \in \text{RV}(E)\} \cup \\ \{(\alpha,\, [\![(R^{\text{rvar}(\alpha,\,E),\,h}/\text{negF}(\alpha,\, E)) \cup \text{posR}(\alpha,\, E)]\!]^{-1}) \mid \alpha \in \text{LV}(E) \end{array}\right)$$

Portanto, pela definição 6.95, $S \in \text{normsol}(E)$. □

Teorema 6.99. $\text{solset}(E) = \text{normsol}(E)$.

Demonstração. Consequência direta dos lemas 6.97 e 6.98. □

Teorema 6.100. *Se E é um sistema de equações $\preceq$-normal tal que* $\text{solset}(E) \neq \emptyset$*, então a substituição identidade é a menor generalização comum (*lcg*) de todas as soluções de E.*

Demonstração. Informalmente, se E é vazio, então id é uma solução de E e, portanto, deve ser o *lcg* das soluções de E, pois é pelo menos tão geral quanto qualquer outra substituição.

Por outro lado, se E é não-vazio, então possui um número infinito de soluções, como descrito na definição 6.95. Podemos escolher então uma função h_a para o lugar da função h da definição 6.95 de forma que o valor correspondente de S_1, que chamaremos S_a, seja tal que $S_a\alpha \neq S_a\beta$ para quaisquer variáveis de registro α e β. Além disso, dado qualquer solução S_b, podemos escolher uma função h_c de maneira que a substituição correspondente S_c tenha a propriedade de que $S_c(\alpha) \neq S_b(\alpha)$ para toda variável de registro α.

A última afirmação significa que se S é o *lcg* das soluções de E, então $S(\alpha)$ tem que ser uma variável de tipo, para toda variável de registro α de E. Da mesma maneira, a penúltima afirmação significa que $S\alpha \neq S\beta$ para todas variáveis de registro $\alpha \neq \beta$ de E.

Portanto, S deve ser uma substituição permutativa e, portanto, isomórfica à substituição identidade, o que significa que a substituição identidade é também um *lcg* das soluções de E. □

6.8 Forma pré-normal

Definição 6.101. Uma equação é considerada uma equação em forma *pré-normal* se possui a seguinte forma:

$$t_1 \equiv (t_2 \ominus R^-) \oplus R^+$$

onde t_1 e t_2 podem ser ou variáveis de registro (r_α, para algum α) ou um conjunto de campos.

Definição 6.102. Um sistema de equações E é um sistema em forma *pré-normal* se todas as suas equações forem pré-normais.

Lema 6.103. *Um sistema de equações $\preceq$-normal é também pré-normal.*

Demonstração. A condição 6.80, necessária para que o sistema seja $\preceq$-normal, especifica que todas as equações do sistema tenham a forma:

$$r_\alpha \equiv (r_\beta \ominus R^-) \oplus R^+$$

Portanto, pela definição 6.101, o sistema é pré-normal. □

Lema 6.104. *Se t é uma variável de registro r_α, para algum α, ou um conjunto de campos R, para algum R, e se S é uma substituição tal que St é definido, então St é ou uma variável de registro, ou um conjunto de campos.*

Demonstração. Suponha que $t = r_\alpha$ para algum α. Pela definição 6.58, St é definido se e somente se $S\alpha$ for ou um tipo registro τ ou uma variável de tipo β. No primeiro caso, temos que $St = [\![\tau]\!]$, que é um conjunto de campos. No segundo caso, temos que $St = r_\beta$, que é uma variável de registro.

Suponha que $t = R$ para algum conjunto de campos R. Pelas definições 6.58 e 6.52, St é definido e é um conjunto de campos. □

Lema 6.105. *Se E é um sistema equações pré-normal, e se S é uma substituição tal que $S(E)$ é definido, então $S(E)$ também é um sistema de equações pré-normal.*

Demonstração. Pelas definições 6.101 e 6.102, cada equação e de E precisa ter a forma:

$$t_1 \equiv (t_2 \ominus R^-) \oplus R^+$$

onde t_1 e t_2 são variáveis de registro ou conjuntos de campos.

Pelas definições 6.64 e 6.58, temos que Se tem a forma:

$$St_1 \equiv (St_2 \ominus R^-_*) \oplus R^+_*$$

onde $R^-_* = S(R^-)$ e $R^+_* = S(R^+)$.

O lema 6.104 nos garante que $S(t_1)$ e $S(t_2)$ são ou variáveis de registro ou conjuntos de campos. Portanto, Se também é pré-normal e, generalizando, temos que $S(E)$ também é pré-normal. □

6.9 O sistema de redução $\mathcal{W}$

Esta seção define o sistema de redução $\mathcal{W} = (\mathbf{P}, \mathbf{V}, \mathbf{R})$, onde $\mathbf{P}$ é o conjunto de instâncias de problemas, definido abaixo, $\mathbf{V}$ é o conjunto de variáveis de tipos e $\mathbf{R}$ é o conjunto de regras de redução definidas adiante nesta seção.

Instâncias de problemas

Instâncias de problemas são os valores sobre os quais o sistema de redução é definido. Formalmente:

Definição 6.106. Uma *instância de problema* $P \in \mathbf{P}$ é o símbolo $\bot$, que representa problemas insolúveis, ou uma tripla (E, S, V), onde E é um sistema de equações pré-normal, S é uma substituição, e V é um conjunto finito de variáveis de tipo.

Definição 6.107. Uma instância de problema $P = (E, S, V)$ é chamada $\preceq$-*normal* se E for um sistema de equações $\preceq$-normal.

Definição 6.108. O conjunto de soluções de uma instância de problema P é dado pela função solset, definida como:

$$\begin{aligned} \mathrm{solset}(\bot) &= \emptyset \\ \mathrm{solset}((E, S, V)) &= \left\{ S' \circ S \mid \begin{array}{l} S' \in \mathrm{solset}(E) \text{ e} \\ S'(\alpha) \text{ é um tipo registro, para } \alpha \in V \end{array} \right\} \end{aligned}$$

Lema 6.109. $\mathrm{solset}(E, \mathrm{id}, \emptyset) = \mathrm{solset}(E)$.

Demonstração.

$$\begin{aligned} \mathrm{solset}((\mathrm{E}, \mathrm{id}, \emptyset)) &= = \left\{ S' \circ \mathrm{id} \mid \begin{array}{l} S' \in \mathrm{solset}(E) \text{ e} \\ S'(\alpha) \text{ é um tipo registro, para } \alpha \in \emptyset \end{array} \right\} \\ &= \{S' \mid S' \in \mathrm{solset}(E)\} \\ &= \mathrm{solset}(E) \end{aligned}$$

□

Definição 6.110. Duas instâncias de problema P_1 e P_2 são consideradas equivalentes com respeito ao seu conjunto de soluções se $\mathrm{solset}(P_1) = \mathrm{solset}(P_2)$. Usamos o símbolo $\simeq$ para denotar esta relação de equivalência.

Definição 6.111. A aplicação de uma substituição à uma instância de problema (E, S, V) é definida como:

$$S'((E, S, V)) = \begin{cases} (S'(E), S' \circ S, S'(V)) & \text{se } S'(E) \text{ é definido, e para todo } \alpha \in V, \\ & S'\alpha \text{ é um tipo registro ou variável} \\ \bot, \text{ caso contrário} & \end{cases}$$

onde

$$S'(V) = \{S'\alpha \mid \alpha \in V,\ S'\alpha \text{ é uma variável de tipo}\}$$

Regras de redução

As seguintes regras de redução definem o sistema de redução sobre o conjunto de instâncias de problemas. Cada regra é definida por: um termo fonte, que representa um padrão que é casado com instâncias de problemas; um conjunto (possivelmente vazio) de pré-condições, que restringe a aplicação da regra; e um conjunto de termos, chamados termos alvo.

Quando uma regra de redução especifica mais de um termo alvo, cada termo alvo será acompanhado de predicados mutuamente exclusivos sobre as variáveis que aparecem no termo fonte.

A primeira regra se distingue das demais por que se aplica apenas à instâncias de problema não-primitivas, enquanto que as outras se aplicam à instâncias primitivas, conforme a definição abaixo:

Definição 6.112. Uma instância de problema é considerada primitiva se e somente se $1 \leq |E| \leq 2$, $S = \mathrm{id}$ e $V = \emptyset$.

Como as regras que tratam de instâncias primitivas só se aplicam a estas, usaremos a notação $E \longrightarrow P$ como abreviação para $(E, \mathrm{id}, \emptyset) \longrightarrow P$.

Decomposição do problema

Regra de Redução 1. Se $E_2 \neq \emptyset$, $1 \leq |E_1| \leq 2$, $(E_1 \cup E_2, S, V)$ é não-primitivo e $(E_1, \mathrm{id}, \emptyset) \longrightarrow P$, então:

$$(E_1 \cup E_2, S, V) \longrightarrow \begin{cases} S'(E_1' \cup E_2, S, V \cup V') & \text{se } P = (E_1', S', V') \\ & \bot, \text{ caso contrário} \end{cases}$$

onde $S'(V) = \{S'\alpha \mid \alpha \in V \text{ e } S'\alpha \text{ é uma variável de tipo}\}$.

Equações que não satisfazem a condição 6.80

Regra de Redução 2. Se X tem a forma r_α ou R', para algum α ou R', então:

$$\{X \equiv (R \ominus R^-) \oplus R^+\} \longrightarrow \begin{cases} (\emptyset, S, \emptyset) & \text{se } \mathrm{fn}(R^-) \subseteq \mathrm{fn}(R), \\ & \quad \mathrm{fn}(R) \cap \mathrm{fn}(R^+) \subseteq \mathrm{fn}(R^-) \\ & \quad \text{e } S \text{ é definida} \\ \bot & \text{, caso contrário} \end{cases}$$

onde

$$S \;=\; \mathrm{mgu}\left(\mathrm{cortypes}(R, R^-) \cup \left\{([\![X]\!]^{-1}, [\![(R/\mathrm{fnames}(R^-)) \cup R^+]\!]^{-1})\right\}\right)$$

Regra de Redução 3.

$$\{R \equiv (r_\alpha \ominus R^-) \oplus R^+\} \longrightarrow \begin{cases} (\emptyset, S, \emptyset) & \text{se } \mathrm{fn}(R^+) \subseteq \mathrm{fn}(R), \\ & \quad \mathrm{fn}(R) \cap \mathrm{fn}(R^-) \subseteq \mathrm{fn}(R^+) \\ & \quad \text{e } S \text{ é definida} \\ \bot & \text{, caso contrário} \end{cases}$$

onde

$$S \;=\; \mathrm{mgu}\left(\mathrm{cortypes}(R,\, R^+) \cup \left\{([\![X]\!]^{-1}, [\![(R/\mathrm{fnames}(R^+)) \cup R^-]\!]^{-1})\right\}\right)$$

Equações que não satisfazem a condição 6.81

Regra de Redução 4. Se $\alpha \prec \beta$ então

$$\{r_\alpha \equiv (r_\beta \ominus R^-) \oplus R^+\} \longrightarrow (\{r_\beta \equiv (r_\alpha \ominus R^+) \oplus R^-\},\, \mathrm{id},\, \emptyset)$$

Equações que não satisfazem a condição 6.82

Regra de Redução 5.

$$\{r_\alpha \equiv (r_\beta \ominus \emptyset) \oplus \emptyset\} \longrightarrow (\emptyset,\, [\alpha \mapsto \beta],\, \{\beta\})$$

Equações que não satisfazem a condição 6.83

Regra de Redução 6. Se $R^- \neq R^+$ e e tem a forma $r_\alpha \equiv (r_\alpha \ominus R^-) \oplus R^+$, para algum α, então:

$$\{r_\alpha \equiv (r_\alpha \ominus R^-) \oplus R^+\} \longrightarrow \begin{cases} S(\{r_\alpha \equiv (r_\alpha \ominus R^-) \oplus R^+\},\, \mathrm{id},\, \emptyset) \\ \qquad \text{se} \quad \mathrm{fn}(R^-) = \mathrm{fn}(R^+) \\ \qquad\qquad \text{e } S \text{ é definida} \\ \bot, \text{ caso contrário} \end{cases}$$

onde

$$S = \mathrm{mgu}(\mathrm{cortypes}(R^-,\, R^+))$$

Regra de Redução 7. Se $\beta \prec \alpha$ então

$$\{r_\alpha \equiv (r_\beta \ominus R) \oplus R\} \longrightarrow S(\{r_\alpha \equiv (r_\beta \ominus R) \oplus R\},\, \mathrm{id},\, \emptyset)$$

onde $S = [\alpha \mapsto \beta]$.

Equações que não satisfazem a condição 6.84

Regra de Redução 8. Se $\beta_2 \preceq \beta_1 \prec \alpha$, então:

$$\left\{ \begin{array}{lcl} r_\alpha & \equiv & (r_{\beta_1} \ominus R_1^-) \oplus R_1^+ \\ r_\alpha & \equiv & (r_{\beta_2} \ominus R_2^-) \oplus R_2^+ \end{array} \right\} \longrightarrow \begin{cases} S(E,\, \mathrm{id},\, \emptyset) \\ \qquad \text{se} \quad \mathrm{fn}(R_2^-) \cap \mathrm{fn}(R_1^+) \subseteq \mathrm{fn}(R_2^+), \\ \qquad\qquad \mathrm{fn}(R_2^+) \cap \mathrm{fn}(R_1^-) \subseteq \mathrm{fn}(R_1^+), \\ \qquad\qquad \text{e } S \text{ é definida} \\ \bot, \text{ caso contrário} \end{cases}$$

onde

$$\begin{array}{rcl} R_3^- & = & R_2^- \cup (R_1^+/\mathrm{fn}(R_2^+)) \\ R_3^+ & = & (R_2^+/\mathrm{fn}(R_1^+)) \cup R_1^- \\ S & = & \mathrm{mgu}(\mathrm{cortypes}(R_1^+,\, R_2^+)) \\ E & = & \left\{ \begin{array}{rcl} r_\alpha & \equiv & (r_{\beta_1} \ominus R_1^-) \oplus R_1^+ \\ r_{\beta_1} & \equiv & (r_{\beta_2} \ominus R_3^-) \oplus R_3^+ \end{array} \right\} \end{array}$$

Regra de Redução 9. Se $\beta \preceq \alpha$ então:

$$\left\{ \begin{array}{rcl} r_\alpha & \equiv & (r_\alpha \ominus R) \oplus R \\ r_\alpha & \equiv & (r_\beta \ominus R^-) \oplus R^+ \end{array} \right\} \longrightarrow \begin{cases} S\left(\left\{r_\alpha \equiv (r_\beta \ominus R_3^-) \oplus R_3^+\right\},\, \mathrm{id},\, \emptyset\right) \\ \qquad \text{se} \;\; \mathrm{fn}(R^-) \cap \mathrm{fn}(R) \subseteq \mathrm{fn}(R^+), \\ \qquad\quad \text{e } S \text{ é definida} \\ \bot, \;\text{caso contrário} \end{cases}$$

onde

$$\begin{array}{rcl} R_3^- & = & R^- \cup (R/\mathrm{fn}(R^+)) \\ R_3^+ & = & (R^+/\mathrm{fn}(R)) \cup R \\ S & = & \mathrm{mgu}(\mathrm{cortypes}(R,\, R^+)) \end{array}$$

Equações que não satisfazem a condição 6.85

Regra de Redução 10. Se $\alpha_1 \prec \alpha_2$ então:

$$\left\{ \begin{array}{rcl} r_{\alpha_1} & \equiv & (r_\beta \ominus R^-) \oplus R^+ \\ r_{\alpha_2} & \equiv & (r_\beta \ominus R^-) \oplus R^+ \end{array} \right\} \longrightarrow \left(S\left(\left\{r_{\alpha_1} \equiv (r_\beta \ominus R^-) \oplus R^+\right\}\right),\, S,\, \emptyset\right)$$

onde

$$S = [\alpha_2 \mapsto \alpha_1]$$

Equações que não satisfazem a condição 6.86

Regra de Redução 11. Se $\alpha_2 \prec \beta \prec \alpha_1$, então:

$$\left\{ \begin{array}{rcl} r_{\alpha_1} & \equiv & (r_\beta \ominus R_1^-) \oplus R_1^+ \\ r_\beta & \equiv & (r_{\alpha_2} \ominus R_2^-) \oplus R_2^+ \end{array} \right\} \longrightarrow \begin{cases} S\,(E,\, \mathrm{id},\, \emptyset) \\ \qquad \text{se} \;\; \mathrm{fn}(R_1^-) \cap \mathrm{fn}(R_2^-) \subseteq \mathrm{fn}(R_2^+), \\ \qquad\quad \mathrm{fn}(R_1^+) \cap \mathrm{fn}(R_2^+) \subseteq \mathrm{fn}(R_1^-), \\ \qquad\quad \text{e } S \text{ é definido} \\ \bot, \quad \text{caso contrário} \end{cases}$$

onde

$$\begin{array}{rcl} S & = & \mathrm{mgu}(\mathrm{cortypes}(R_1^-,\, R_2^+)) \\ R_3^- & = & R_2^- \cup (R_1^- / \mathrm{fn}(R_2^+)) \\ R_3^+ & = & (R_2^+ / \mathrm{fn}(R_1^-)) \cup R_1^+ \\ E & = & \left\{ \begin{array}{rcl} r_{\alpha_1} & \equiv & (r_{\alpha_2} \ominus R_3^-) \oplus R_3^+ \\ r_\beta & \equiv & (r_{\alpha_2} \ominus R_2^-) \oplus R_2^+ \end{array} \right\} \end{array}$$

Regra de Redução 12. Se $\beta \preceq \alpha$ então:

$$\left\{ \begin{array}{rcl} r_\alpha & \equiv & (r_\beta \ominus R^-) \oplus R^+ \\ r_\beta & \equiv & (r_\beta \ominus R) \oplus R \end{array} \right\} \longrightarrow \begin{cases} S(\{r_\alpha \equiv (r_\beta \ominus R_1^-) \oplus R_1^+\},\, \mathrm{id},\, \emptyset) \\ \qquad \text{se} \quad \mathrm{fn}(R^-) \cap \mathrm{fn}(R) \subseteq \mathrm{fn}(R^+), \\ \qquad\quad \mathrm{fn}(R^+) \cap \mathrm{fn}(R) \subseteq \mathrm{fn}(R^-), \\ \qquad\quad \text{e } S \text{ é definido} \\ \bot, \quad \text{caso contrário} \end{cases}$$

onde

$$\begin{array}{rcl} R_1^- & = & R^- \cup (R / \mathrm{fn}(R^+)) \\ R_1^+ & = & (R^+ / \mathrm{fn}(R)) \cup R \\ S & = & \mathrm{mgu}(\mathrm{cortypes}(R^+,\, R)) \end{array}$$

Equações que não satisfazem a condição 6.87

Regra de Redução 13. Se $\beta \preceq \alpha_1$, $\beta \preceq \alpha_2$, $\mathrm{cortypes}(R_1^-, R_2^-) \not\subseteq \Delta\mathbf{T}$ e

$$E = \left\{ \begin{array}{rcl} r_{\alpha_1} & \equiv & (r_\beta \ominus R_1^-) \oplus R_1^+ \\ r_{\alpha_2} & \equiv & (r_\beta \ominus R_2^-) \oplus R_2^+ \end{array} \right\}$$

então:

$$E \longrightarrow \begin{cases} S(E,\, \mathrm{id},\, \emptyset) & \text{se} \quad S \text{ é definido} \\ \bot & \text{caso contrário} \end{cases}$$

onde

$$S = \mathrm{mgu}(\mathrm{cortypes}(R_1^-,\, R_2^-))$$

Equações que não satisfazem a condição 6.88

Regra de Redução 14. Se $\mathrm{fn}(R_1^-) \cap \mathrm{fn}(R_2^+) \not\subseteq \mathrm{fn}(R_2^-)$, então:

$$\left\{ \begin{array}{rcl} r_{\alpha_1} & \equiv & (r_\beta \ominus R_1^-) \oplus R_1^+ \\ r_{\alpha_2} & \equiv & (r_\beta \ominus R_2^-) \oplus R_2^+ \end{array} \right\} \longrightarrow \bot$$

Teoremas sobre o sistema de redução

Definição 6.113. Se P e P' são instâncias de problemas, dizemos que $P \xrightarrow{\mathcal{W}} P'$, ou simplesmente que $P \longrightarrow P'$, se existe uma regra do sistema $\mathcal{W}$ que reduz P a P'. Se não existe um P' tal que $P \longrightarrow P'$, dizemos que P' é um problema $\mathcal{W}$-normal.

Dizemos que $P \xrightarrow{*} P'$ se $P = P'$ ou se P reduz a P' através de um ou mais passos. Dizemos também que $P \xrightarrow{!} P'$ se $P \xrightarrow{*} P'$ e P' é uma forma $\mathcal{W}$-normal.

Lema 6.114. $(\mathbf{P}, \xrightarrow{*})$ *é um conjunto pré-ordenado.*

Demonstração. Por sua definição, $\xrightarrow{*}$ é o fechamento transitivo e reflexivo da relação $\longrightarrow$ e, portanto, uma pré-ordem. □

Teorema 6.115. *O conjunto de instâncias de problemas é fechado sob as regras de redução de $\mathcal{W}$.*

Demonstração. Sejam $P = (E, S, V)$ e P' instâncias de problemas tais que $P \longrightarrow P'$.

Suponha primeiramente que P seja primitivo, isto é, $P = (E, \mathrm{id}, \emptyset)$ onde $1 \leq |E| \leq 2$. A regra de decomposição (regra 1) não pode ser aplicada a P. Cada uma das outras regras ou reduzem P a $\perp$ ou fazem uma ou mais das seguintes operações:

- remove uma equação de E;
- altera uma ou duas equações de E;
- aplica uma substituição ao resultado das outras operações.

A remoção de uma equação de um sistema pré-normal resulta em sistema pré-normal.

Quando uma equação altera uma ou mais equações de E, as equações resultantes também são pré-normais, como pode ser verificado pelas definições das regras de redução.

O resultado de se aplicar uma substituição a um sistema pré-normal é ou o problema insolúvel ($\perp$) ou então, pelo lema 6.105, o resultado é também um sistema pré-normal. □

Teorema 6.116. *Se $P = (E, S, V)$ é uma instância de problema tal que E não é um sistema de equação $\preceq$-normal, então existe um P' tal que $P \xrightarrow{\mathcal{W}} P'$.*

Demonstração. Se E não é $\preceq$-normal, então não satisfaz uma das equações descritas na definição 6.79. Fazendo uma inspeção das condições especificadas em cada regra de redução, podemos verificar que se uma das condições descritas em 6.79 não for satisfeita, então pelo menos uma das regras de redução será aplicável. □

Corolário 6.117. *Se P é uma instância de problema $\mathcal{W}$-normal, então P é $\perp$ ou uma instância de problema $\preceq$-normal.*

Definição 6.118. Um sistema de redução $\mathcal{R}$ é considerado um sistema *terminante* se não existirem sequências infinitas de redução $X_1 \xrightarrow{\mathcal{R}} X_2 \xrightarrow{\mathcal{R}} \cdots$.

Teorema 6.119. *O sistema de redução $\mathcal{W}$ é terminante.*

Demonstração. Como a prova deste teorema é longa, seus detalhes serão apresentado na próxima seção. Faremos aqui apenas um resumo de sua estrutura.

A estratégia para provar que $\mathcal{W}$ é terminante é a construção de uma ordem total bem fundada $(\mathbf{A}, \leq_{\mathbf{A}})$ e uma função decrescente f da pré-ordem $(\mathbf{P}, \xrightarrow{*})$ para $(\mathbf{A}, \leq_{\mathbf{A}})$.

Suponha que $\mathbb{P} \subseteq \mathbf{P}$ é conjunto de instâncias de problemas limitado inferiormente, de forma que $\mathbb{P}$ forma uma cadeia de redução de $\mathcal{W}$. Seja

$$\mathbb{A} = \{a \in \mathbf{A} \mid a = f(P), \text{ para algum } P \in \mathbb{P}\}$$

a imagem de f sobre $\mathbb{P}$.

Como sabemos que f é decrescente, e que $\mathbb{P}$ é limitado inferiormente, então $\mathbb{A}$ deve ser limitado superiormente. E como $\mathbb{P}$ forma uma cadeia, então $\mathbb{A}$ também forma uma cadeia.

Já que $(\mathbf{A}, \leq_{\mathbf{A}})$ é bem fundada, então $\mathbb{A}$ tem que ser finito.

f é uma função decrescente e, portanto, injetiva, o que significa que $|\mathbb{P}| = |\mathbb{A}|$. Logo, $\mathbb{P}$ têm de ser finito e, portanto, não há cadeias infinitas de redução no sistema $\mathcal{W}$. □

Definição 6.120. A função $f : \mathbf{P} \to \mathbf{Y}$ é considerada uma função $\mathcal{W}$-invariante se para todo P e P' tais que $P \xrightarrow{*} P'$, tivermos que $f(P) = f(P')$.

Lema 6.121. *f é $\mathcal{W}$-invariante se e somente se, para todo P e P' tais que $P \longrightarrow P'$, tivermos que $f(P) = f(P')$.*

Demonstração. Pela definição 6.120, se f é $\mathcal{W}$-invariante, então o seu valor é preservado por uma quantidade qualquer de passos de redução. Em particular, se $P \longrightarrow P'$ então $P \xrightarrow{*} P'$ e portanto $f(P) = f(P')$.

Por outro lado, se $f(P) = f(P')$ para todo P e P' tais que $P \longrightarrow P'$, então para toda cadeia

$$P \to P_1 \longrightarrow \cdots \longrightarrow P_n \to P'$$

temos que:

$$f(P_0) = f(P_1) = \cdots f(P_n)$$

Portanto, se $P \xrightarrow{*} P'$, então $f(P) = f(P')$, o que significa que f é um $\mathcal{W}$-invariante. □

Teorema 6.122. *A função* solset *é uma função $\mathcal{W}$-invariante.*

Demonstração. Este teorema pode ser provado demonstrando-se que todas as regras de redução preservam o conjunto de soluções das instâncias de problema.

Cada regra r_i, para $1 \leq i \leq 13$, consiste em uma condição c_i, um termo fonte p_i^h, e um conjunto de termos alvo associados à condições $\{(c_{i,n}, p_{i,n})\}$. Para cada regra i e para cada par $(c_{i,n}, p_{i,n})$ da regra, temos que provar que, sob as condições c_i e $c_{i,n}$, se S é uma solução para a instância de problema com forma p_i, então também é uma solução para $p_{i,n}$, e vice-versa.

A prova completa deste teorema será omitida deste documento. □

Teorema 6.123. *Se E é um sistema de equações pré-normal, e $(E, \mathrm{id}, \emptyset) \xrightarrow{!} (E', S, V)$, então S é uma menor generalização comum de todas as soluções de E. Formalmente, i.e. $S = \bigvee \mathrm{solset}(E)$.*

Demonstração. Pelo lema 6.109, pelo corolário 6.117 e pelo teorema 6.122, temos:

$$\begin{aligned}
\mathrm{solset}(E) &= \mathrm{solset}(E, \mathrm{id}, \emptyset) \\
&= \mathrm{solset}(E', S, V) \\
&= \left\{ S' \circ S \mid \begin{array}{l} S' \in \mathrm{solset}(E') \text{ e} \\ \quad S'\alpha \text{ é um tipo registro, para } \alpha \in \mathrm{V} \end{array} \right\}
\end{aligned}$$

Uma propriedade útil de uma menor generalização comum de um conjunto de substituições é que se $\mathbb{S}_1$ e $\mathbb{S}_2$ são conjuntos não-vazios de substituições e se S é uma substituição tal que $\mathbb{S}_1 = \{S' \circ S \mid S' \in \mathbb{S}_2\}$, então $\bigvee \mathbb{S}_1 = \bigvee \mathbb{S}_2 \circ S$.

Sabemos também, pela definição 6.95, que para toda variável α, o conjunto de soluções de E' ou restringe α de forma que α só possa ser instanciado para elementos de um conjunto infinito de tipos registro, ou α não é restringido de nenhuma forma. Portanto, se $\alpha \in V$, então α é restringido por E' e, portanto, a condição

$$S'\alpha \text{ é um tipo registro para } \alpha \in V$$

é satisfeita para todo $S' \in \mathrm{solset}(E)$.

Temos então que:

$$\begin{aligned}
\bigvee \mathrm{solset}(E) &= \bigvee \left(\left\{ S' \circ S \mid \begin{array}{l} S' \in \mathrm{solset}(E') \text{ e} \\ \quad S'\alpha \text{ é um tipo registro, para } \alpha \in \mathrm{V} \end{array} \right\} \right) \\
&= \bigvee \left(\{ \; S' \circ S \mid \; S' \in \mathrm{solset}(E') \text{ e } \} \right) \\
&= \bigvee \mathrm{solset}(E') \circ S
\end{aligned}$$

Pelo teorema 6.100, sabemos que $\bigvee \mathrm{solset}(E')$ é a substituição identidade, ou isomorfo à mesma, e portanto, podemos concluir que:

$$\bigvee \mathrm{solset}(E) = S$$

□

6.10 Propriedades de terminação do sistema de redução

Para provar a propriedade de terminação do sistema de redução $(\mathbf{P}, \xrightarrow{\mathcal{W}})$, construiremos um conjunto bem fundado $(R, \prec_R)$, que chamaremos de conjunto ordenado de referência,

e uma função $\mathrm{k} : \mathbf{P} \mapsto R$ estritamente decrescente, ou seja, tal que para todo p e p', se $p \xrightarrow{\mathcal{W}} p'$ então $\mathrm{k}(p') \prec_R \mathrm{k}(p)$.

Se $\xrightarrow{\mathcal{W}}$ fosse não-terminante, teria de haver ao menos uma cadeia infinita de redução:

$$p_1 \longrightarrow p_2 \longrightarrow \cdots \longrightarrow p_i \longrightarrow \cdots$$

Porém, isto significaria que teria de haver também uma cadeia decrescente infinita em R:

$$\mathrm{k}(p_1) \succ_R \mathrm{k}(p_2) \succ \cdots \succ_R \mathrm{k}(p_i) \succ \cdots$$

Como $\prec_R$ é bem fundada, tal cadeia não pode existir e, portanto, não podem haver cadeias de redução infinitas em $(\mathbf{P}, \xrightarrow{\mathcal{W}})$. Com isto estabelecemos que o sistema de redução é terminante.

Para evitar confusão, usaremos o símbolo $\preceq_{\mathbf{V}}$ para representar a ordem total entre variáveis de tipo que são usadas na definição da forma $\preceq$-normal e nas regras de redução. Também assumiremos que as variáveis de tipo podem ser enumeradas por uma função $\mathrm{ord} : \mathbf{V} \to \mathbb{N}$, onde

$$\mathrm{se} \alpha_1 \prec_{\mathbf{V}} \alpha_2 \text{ então } \mathrm{ord}(\alpha_1) < \mathrm{ord}(\alpha_2)$$

Chamamos o valor $\mathrm{ord}(\alpha)$ de *número de ordem de* α.

Usaremos o conjunto $(\mathbb{N}^4, \leq^*)$ como o conjunto ordenado de referência, onde $\mathbb{N}^4$ é o conjunto de quádruplas de números naturais e $\leq^*$ é a ordem lexicográfica entre elementos de $\mathbb{N}^4$. Descreveremos elementos de $\mathbb{N}^4$ usando notação vetorial, como em $\langle 1, 4, 10, 7 \rangle$.

A relação $\leq^*$ é regida pelas seguintes propriedades:

- Se $x <^* x'$, então $\langle x, y, z, w \rangle <^* \langle x', y', z', w' \rangle$.
- Se $x = x'$ e $y < y'$, então $\langle x, y, z, w \rangle <^* \langle x', y', z', w' \rangle$.
- Se $x = x'$, $y = y'$ e $z < z'$, então $\langle x, y, z, w \rangle <^* \langle x', y', z', w' \rangle$.
- Se $x = x'$, $y = y'$, $z = z'$ e $w < w'$, então $\langle x, y, z, w \rangle <^* \langle x', y', z', w' \rangle$.

Antes de definir a função k, definiremos algumas funções auxiliares:

Definição 6.124. O número de equações em uma instância de problema p é dado pela função $\mathrm{eqcount} : \mathbf{P} \to \mathbb{N}$, onde:

$$\mathrm{eqcount}((E, S, V)) = |E|$$

Definição 6.125. O número de variáveis de tipo que ocorrem em uma instância de problema p é dado pela função $\mathrm{varcount} : \mathbf{P} \to \mathbb{N}$, definida como:

$$\mathrm{varcount}((E, S, V)) \quad = \quad |\{\alpha \,|\, \alpha \in \mathrm{tv}(e) \text{ e } e \in E\}|$$

Definição 6.126. O número de equações em uma instância de problema cujo termo esquerdo é a variável de registro r_α é dado pela função lvareqs : $\mathbf{V} \times \mathbf{P} \to \mathbb{N}$, definida como:

$$\text{lvarseq}(\alpha,\, (E,\, S,\, V)) = \left| \left\{ \begin{array}{ll} e \,| & e \in E \text{ e} \\ & e = r_\alpha \equiv (r_{\beta_1} \ominus R_1^-) \oplus R_1^+ \end{array} \right\} \right|$$

Definição 6.127. O *índice de repetição à esquerda* de uma instância de problema p é dado pela função lrepindex : $\mathbf{P} \to \mathbb{N}$, definida como:

$$\text{lrepindex}((E,\, S,\, V)) = \sum_{\alpha \in \text{tv}(E)} \text{ord}(\alpha) \cdot \text{lvarseq}(\alpha,\, (E,\, S,\, V))$$

Definição 6.128. O *número de ordem* de uma instância de problema p é dado pela função $\text{ord}(p)$, definida como o maior *número de ordem* de uma variável que ocorre em p:

$$\text{ord}(p) = \max(\{\text{ord}(\alpha) \,|\, \alpha \in \text{tv}(p)\})$$

Definição 6.129. Dada uma instância de problema p, a *largura* de uma equação $e \in p$ com a forma $r_\alpha \equiv (r_\beta \ominus R^-) \oplus R^+$ é dada pela função span(e, p), definida como:

$$\text{span}(e,\, p) = \begin{cases} \text{ord}(p) & \text{se } \alpha \preceq_{\mathbf{V}} \beta \\ \text{ord}(p) - (\text{ord}(\alpha) - \text{ord}(\beta)) & \text{se } \beta \prec_{\mathbf{V}} \alpha \end{cases}$$

Para ver que $\text{span}(e,\, p)$ é bem definida, temos de recordar que $\text{ord}(p) \geq \text{ord}(\alpha)$, para toda variável de registro α em p. Portanto, temos que $\text{ord}(p) \geq \text{ord}(\alpha)$ na equação acima. Como $\beta \prec_{\mathbf{V}} \alpha$, então $\text{ord}(\beta) < \text{ord}(\alpha)$ e portanto:

$$0 \leq \text{ord}(\alpha) - \text{ord}(\beta) \leq \text{ord}(\alpha)$$

Por isso temos:

$$\text{ord}(p) \geq \text{ord}(\alpha) - \text{ord}(\beta)$$

e

$$\text{span}(e,\, p) = \text{ord}(p) - (\text{ord}(\alpha) - \text{ord}(\beta)) \geq 0$$

Definição 6.130. A *largura* de uma instância de problema p é a soma de todas as larguras de todas as equações de p:

$$\text{span}((E,\, S,\, V)) = \sum \{\text{span}(e,\, (E,\, S,\, V)) \,|\, e \in E \text{ e } e = r_\alpha \equiv (r_\beta \ominus R^-) \oplus R^+\}$$

A função k nos dá, para cada instância de problema p, o número de equações em p, o número de variáveis de tipo em p, o índice de repetição à esquerda de p e a largura de p:

$$\text{k}(p) = \begin{cases} \langle \text{eqcount}(p),\, \text{varcount}(p),\, \text{lrepindex}(p),\, \text{span}(p) \rangle & \text{se } p = (E,\, S) \\ \langle 0,\, 0,\, 0,\, 0 \rangle & \text{se } p = \bot \end{cases}$$

Dizemos que uma regra de redução de $\mathcal{W}$ é $\leq^*$-decrescente se para todo p e p', se $p \longrightarrow p'$ pela regra, então $\text{k}(p') \leq^* \text{k}(p)$.

Os seguintes lemas provam que todas as regras de $\mathcal{W}$ são $\leq^*$-decrescentes.

Lema 6.131. *As regras de redução 2, 3, 9, 10 e 12 são $\leq^*$-decrescentes.*

Demonstração. Estas regras de redução reduzem uma instância de problema a $\bot$ ou reduzem o sistema de equações correspondente ao conjunto vazio (2 e 3), ou reduzem um sistema de equações com duas equações para um sistema com apenas uma equação (9, 10 e 12).

No primeiro caso, como uma instância com pelo menos uma equação foi reduzida a $\bot$, então temos que a redução é $\leq^*$-decrescente. Nos outros casos, temos uma redução $(E, S, V) \longrightarrow (E', S', V')$, com $\text{eqcount}(E) > \text{eqcount}(E')$. Portanto, pela definição de $\leq^*$ e k, temos que $\text{k}((E, S, V)) > \text{k}((E', S', V'))$. □

Lema 6.132. *Se τ e τ' são unificáveis, então existe um unificador mais geral idempotente de τ e τ'. Além disso, se $\tau \neq \tau'$ então $\text{tv}(S\tau) \subset \text{tv}(\tau) \cup \text{tv}(\tau')$.*

Demonstração. Este é um resultado conhecido sobre unificação. O algoritmo usual para computar o mgu de dois termos nos dá uma substituição idempotente com a propriedade de que se não for a substituição identidade, então o conjunto de variáveis em seu contradomínio é um subconjunto próprio do conjunto de variáveis em seu domínio.

Para um tratamento mais detalhado sobre unificação e substituições, veja [33, 9, 13].

Nos lemas que se seguem, usamos a expressão $S = \text{mgu}(X)$ para denotar que S é um unificador mais geral idempotente do conjunto de pares de tipos X. □

Lema 6.133. *Se τ e τ' são subtermos de uma equação e tais que $\tau \neq \tau'$, e $S = \text{mgu}(\tau, \tau')$, então $\text{tv}(Se) \subset \text{tv}(e)$.*

Demonstração. Se $\tau \neq \tau'$, então pelo menos uma variável presente está presente no domínio de S, caso contrário teríamos que $S = \text{id}$ e $\tau = \tau'$.

Como S não introduz nenhuma variável nova, pois as variáveis de seu contra-domínio são um subconjunto próprio das variáveis de seu domínio, então Se tem pelo menos uma variável a menos que e.

□

Lema 6.134. *A regra de redução 7 é $\leq^*$-decrescente.*

Demonstração. Esta regra reduz uma instância de problema p ao resultado da aplicação de uma substituição $[\alpha \mapsto \beta]$ a p, onde $\alpha \neq \beta$ e $\{\alpha, \beta\} \subseteq \text{tv}(p)$. Como esta aplicação de substituição é válida (veja a definição 6.111), então a instância de problema resultante será idêntica a p, porém com as ocorrências de α substituídas por β.

Portanto, o número de equações na instância resultante é menor ou igual ao número de equações de p, e o número de variáveis é menor do que p.

Portanto, pela definição de k e de $\leq^*$, se p reduz a p' por esta regra, então $\text{k}(p) >^* \text{k}(p')$. □

Lema 6.135. *A regra de redução 14 é $\leq^*$-decrescente.*

Demonstração. Esta regra de redução simplesmente reduz uma instância de problema não-vazia p (eqcount$(p) \geq 1$) para a instância insolúvel ($\bot$). Como $\mathrm{k}(p) \geq \langle 1, 0, 0, 0\rangle > \langle 0, 0, 0, 0\rangle = \mathrm{k}(\bot)$, então $\mathrm{k}(p) >^* \mathrm{k}(\bot)$ e, portanto, esta regra é $\leq^*$-decrescente. □

Lema 6.136. *As regras de redução 6 e 13 são $\leq^*$-decrescentes.*

Demonstração. Estas regras se assemelham em que o resultado de suas aplicações a uma instância de problema p depende da existência de uma substituição S que seja um unificador mais geral de um conjunto de pares de tipos que são subtermos de equações de p, onde pelo menos um destes pares é distinto.

Se tal substituição existir, então a instância de problema resultante p' é igual a $S(p)$. Sendo este o caso, p' teria no máximo o mesmo número de equações que p tem. Como ao menos um dos pares unificados por S é distinto, temos pelo lema 6.133 que $\mathrm{tv}(p') \subset \mathrm{tv}(p)$.

Portanto, pela definição de k e de $\leq^*$, temos que $\mathrm{k}(p) \geq^* \mathrm{k}(p')$ e que estas regras são $\leq^*$-decrescentes. □

Lema 6.137. *A regra de redução 8 é $\leq^*$-decrescente.*

Demonstração. Suponha que p reduz ap' por esta regra. O valor de p' depende da existência e do valor de uma substituição S, que é um unificador mais geral de alguns subtermos de equações de p. Podem haver três casos para o valor de p':

1. $p' = \bot$, se S não existir. Como o conjunto de equações de p é não vazio, temos que $\mathrm{k}(p) \geq^* \mathrm{k}(p')$, como no lema 6.135.

2. $p' = (E', \mathrm{id}, \emptyset)$, onde $p = (E, \mathrm{id}, \emptyset)$, se S existir e for a substituição identidade. Neste caso, E' tem o mesmo número de equações e o mesmo conjunto de variáveis que E tem, porém, uma das equações com uma variável à esquerda α foi substituída por um equação com uma variável à esquerda β_1, onde $\beta_1 <_{\mathbf{V}} \alpha$. Portanto, temos que $\mathrm{lrepindex}(p) = \mathrm{lrepindex}(p') - \mathrm{ord}(\alpha) + \mathrm{ord}(\beta_1)$. Como $\mathrm{ord}(\beta_1) < \mathrm{ord}(\alpha)$, então $\mathrm{lrepindex}(p) > \mathrm{lrepindex}(p')$ e $\mathrm{k}(p) >^* \mathrm{k}(p')$.

3. $p' = (S(E'), S, \emptyset)$, onde $p = (E, \mathrm{id}, \emptyset)$, se S existir e não for a substituição identidade. Neste caso, p' tem no máximo o mesmo número de equações que p e pelo menos uma variável a menos. Portanto, $\mathrm{k}(p) >^* \mathrm{k}(p')$.

Podemos concluir então que a regra 8 é $\leq^*$-decrescente. □

Lema 6.138. *A regra de redução 11 é $\leq^*$-decrescente.*

Demonstração. Suponha que p reduz a p' por esta regra. Novamente, temos que o valor de p' depende da existência e do valor de uma substituição S, que é um unificador mais geral de um certo conjunto de pares de subtermos de equações de p. Temos então, como no lema anterior, três casos para o resultado da redução, sendo que dois dos casos tem as mesmas propriedades dos casos 1 e 3 do lema anterior.

O outro caso ocorre quando a substituição S existe e é igual à substituição identidade. Nesta situação, p' difere de p em que uma equação e com uma variável à esquerda α_1 e uma

variável à direita β é substituída por uma equação e' com a mesma variável à esquerda, porém com a variável à direita igual a α_2, onde $\alpha_2 <_{\mathbf{V}} \beta <_{\mathbf{V}} \alpha_1$.

Quanto às propriedades de p e p': o número de equações de p' é menor ou igual ao de p; o conjunto de variáveis de p é igual ao de p'; o índice de repetição à esquerda de p é igual ao de p', pois as variáveis à esquerda são as mesmas de p e $\mathrm{ord}(p) = \mathrm{ord}(p')$. A diferença na *largura* de p e p' será mostrada a seguir:

Se $p = (E \cup \{e\},\, S)$ e $p' = (E \cup \{e'\},\, S)$, temos:

$$\begin{aligned}
\mathrm{span}((E \cup \{e\},\, S)) &= \mathrm{span}(E,\, S) + \mathrm{span}(e,\, p) \\
&= \mathrm{span}(E,\, S) + \mathrm{ord}(p) - (\mathrm{ord}(\alpha_1) - \mathrm{ord}(\beta)) \\
&= \mathrm{span}(E,\, S) + \mathrm{ord}(p) - \mathrm{ord}(\alpha_1) + \mathrm{ord}(\beta) \\
&> \mathrm{span}(E,\, S) + \mathrm{ord}(p) - \mathrm{ord}(\alpha_1) + \mathrm{ord}(\alpha_2) \\
&= \mathrm{span}(E,\, S) + \mathrm{ord}(p) - (\mathrm{ord}(\alpha_1) - \mathrm{ord}(\alpha_2)) \\
&= \mathrm{span}(E,\, S) + \mathrm{span}(e',\, p) \\
&= \mathrm{span}((E \cup \{e'\},\, S)
\end{aligned}$$

Como p tem o mesmo número de equações e variáveis de tipo que p' e mesmo índice de repetição à esquerda, mas tem um valor mais alto para a função span, então temos que $\mathrm{k}(p) >^* \mathrm{k}(p')$ e, portanto, esta regra é $\leq^*$-decrescente. □

Lema 6.139. *A regra de redução 1 é $\leq^*$-decrescente.*

Demonstração. Se p reduz a p' pela regra 1, então p é não-vazio e p' é igual a $\bot$ ou a $S'(E'_1 \cup E_2,\, S,\, V \cup V')$, onde:

$$\begin{gathered}
p = (E_1 \cup E_2,\, S,\, V) \\
1 \leq |E_1| \leq 2 \\
(E_1,\, \mathrm{id},\, \emptyset) \longrightarrow (E'_1,\, S',\, V')
\end{gathered}$$

Como $1 \leq |E_1| \leq 2$, $(E_1,\, \mathrm{id},\, \emptyset)$ é uma instância primitiva e, portanto, se for redutível, o será pelas regras tratadas pelos lemas anteriores. Podemos supor então que, caso seja redutível, será o caso que:

$$\mathrm{k}((E_1,\, \mathrm{id},\, \emptyset)) >^* \mathrm{k}((E'_1,\, S',\, V'))$$

Além disso, sabemos pelos lemas anteriores que uma das alternativas a seguir deve ser o caso: □

- o número de equações foi reduzido ($|E_1| > |E'_1|$). Neste caso, o número de equações também é reduzido quando da redução de p para p':

  $$|E_1 \cup E_2| \geq |E_1| + |E_2| > |E'_1| + |E_2| \geq |E'_1 \cup E_2|$$

  Portanto, $\mathrm{k}(p) >^* \mathrm{k}(p')$ neste caso.

- o número de equações mantém-se o mesmo, mas o número de variáveis é reduzido ($\text{varcount}(E_1) > \text{varcount}(E'_1)$). Esta situação só ocorre quando $S'(\alpha) \neq \alpha$, para algum $\alpha \in \text{tv}(E_1)$, em cujo caso $\alpha \notin \text{tv}(E'_1)$

  - Se $\alpha \in \text{tv}(E_2)$, então $\text{varcount}(E_2) > \text{varcount}(S'(E_2))$ e, portanto:

    $$\text{varcount}(E_1 \cup E_2) > \text{varcount}(S'(E'_1 \cup E_2))$$

    .
  - Se $\alpha \notin \text{tv}(E_2)$, então $\alpha \notin \text{tv}(S(E'_1 \cup E_2))$ e, portanto:

    $$\text{varcount}(E_2) > \text{varcount}(S'(E_2))$$

    .

  Em ambos os casos, temos que $\text{k}(p) >^* \text{k}(p')$.

- o número de equações mantém-se o mesmo, assim como o número de variáveis, porém $\text{lrepindex}(p) > \text{lrepindex}(p')$ e $S' = \text{id}$.

  Esta situação ocorre quando uma equação com uma variável à esquerda r_{α_1} é substituída por uma equação com uma variável à esquerda r_{α_2}, com $ord(\alpha_1) > \text{ord}(\alpha_2)$. Portanto, $S(E'_1 \cup E_2) = E'_1 \cup E_2$, que difere de $E_1 \cup E_2$ quanto à substituição da equação referida.

  Neste caso, temos que $\text{lrepindex}(E_1 \cup E_2) > \text{lrepindex}(S(E'_1 \cup E_2))$ e, portanto, $\text{k}(p) >^* \text{k}(p')$.

- o número de equações, de variáveis e o índice de repetição à esquerda se mantém os mesmos, mas $\text{span}(p) > \text{span}(p')$ e $S' = \text{id}$.

  A função *span* é definida como:

  $$\text{span}((E,\, S,\, V)) = \sum\{\text{span}(e,\, p) \,|\, e \in E \text{ e } e = r_\alpha \equiv (r_\beta \ominus R^-) \oplus R^+\}$$

  Fazendo $E = E'_1 \cup E_2 = S'(E'_1 \cup E_2)$, temos:

  $$\begin{aligned}
  \text{span}((E'_1 \cup E_2,\, S,\, V)) = &\sum\{\text{span}(e,\, p) \,|\, e \in E_2 \text{ e } e = r_\alpha \equiv (r_\beta \ominus R^-) \oplus R^+\} + \\
  &\sum\{\text{span}(e,\, p) \,|\, e \in (E'_1 - E_2) \text{ e } e = r_\alpha \equiv (r_\beta \ominus R^-) \oplus R^+\} \\
  \leq &\sum\{\text{span}(e,\, p) \,|\, e \in E_2 \text{ e } e = r_\alpha \equiv (r_\beta \ominus R^-) \oplus R^+\} + \\
  &\sum\{\text{span}(e,\, p) \,|\, e \in E'_1 \text{ e } e = r_\alpha \equiv (r_\beta \ominus R^-) \oplus R^+\} \\
  < &\sum\{\text{span}(e,\, p) \,|\, e \in E_2 \text{ e } e = r_\alpha \equiv (r_\beta \ominus R^-) \oplus R^+\} + \\
  &\sum\{\text{span}(e,\, p) \,|\, e \in E_1 \text{ e } e = r_\alpha \equiv (r_\beta \ominus R^-) \oplus R^+\} \\
  = &\text{span}((E_1 \cup E_2,\, S,\, V))
  \end{aligned}$$

Como as regras de redução não mudam as variáveis à esquerda de E_1, temos que $\text{lrepindex}(E_1 \cup E_2) \geq \text{lrepindex}(E_1' \cup E_2)$.

Portanto, temos:

$$\begin{aligned}\text{eqcount}(E_1 \cup E_2) &= \text{eqcount}(E_1' \cup E_2)\\ \text{varcount}(E_1 \cup E_2) &= \text{varcount}(E_1' \cup E_2)\\ \text{lrepindex}(E_1 \cup E_2) &\geq \text{lrepindex}(E_1' \cup E_2)\\ \text{span}(E_1 \cup E_2) &> \text{span}(E_1' \cup E_2)\end{aligned}$$

e podemos então concluir que $\text{k}(p) >^* \text{k}(p')$.

Teorema 6.140. *O sistema de redução $\mathcal{W}$ é terminante.*

Demonstração. Os lemas desta seção demonstraram que todas as regras de $\mathcal{W}$ são $\leq^*$-decrescentes. Como $\leq^*$ é bem fundada, então $\mathcal{W}$ não pode ter cadeias de redução infinitas. □

6.11 De conjuntos de restrições a instâncias de problemas

Dada uma ordem total $\preceq$ sobre variáveis de tipo, cada conjunto de restrições κ está relacionado com um sistema de equações em um contexto Γ mapeando cada uma das restrições a zero ou uma equação conforme se segue, usando variáveis novas quando necessário:

- uma restrição com a forma $+f : \tau_1 \to \tau_2 \to \tau_3$ é mapeada para a equação

  $$[\![\tau_3]\!] \equiv ([\![\tau_2]\!] \ominus \emptyset) \oplus \{(f, \tau_1)\}$$

  se $[\![\tau_2]\!]$ e $[\![\tau_3]\!]$ forem definidos. Caso contrário, a instância de problema resultante será a instância insolúvel ($\bot$).

- uma restrição com a forma $-f : \tau$ é mapeada para a equação

  $$r_{\alpha_1} \equiv ([\![\tau]\!] \ominus \emptyset) \oplus \{(f, \alpha_2)\}$$

  se $[\![\tau]\!]$ for definido, onde α_1 e α_2 são variáveis novas, que não ocorram livres nem em Γ, κ ou em outras equações. Caso contrário, a instância de problema resultante será $\bot$.

Os casos acima em que o conjunto de restrições é mapeado para $\bot$ correspondem ao fato de que os tipos τ_2 e τ_3, na primeira regra, e τ, na segunda regra, só podem ser variáveis de tipo ou tipos registro. Caso o conjunto de restrições possua uma restrição em que está condição não é atendida, dizemos que o conjunto de restrições é *trivialmente insolúvel*.

Dizemos que um sistema equações $\preceq$-normal E é *recursivo* se existir uma variável $\alpha \in \text{tv}(E)$ tal que $\alpha \in \text{subvars}(\alpha, E)$ (veja a definição 6.91).

Formalmente, o mapeamento de conjuntos de restrições a sistemas de equações é dado pela função $\text{probinstance}(\kappa, \Gamma, \preceq)$, definida como:

Definição 6.141.

$$\text{probinstance}(\kappa,\, \Gamma,\, \preceq) = \begin{cases} \bot, \text{ se } \kappa \text{ for trivialmente insolúvel} \\ (\text{eqs}(\kappa,\, \text{tv}(\Gamma) \cup \text{tv}(\kappa),\, \preceq),\, \text{id},\, \emptyset) \\ \qquad \text{caso contrário} \end{cases}$$

$$\begin{aligned} \text{eqs}(\kappa \cup \{c\},\, V,\, \preceq) &= E \cup \text{eqs}(\kappa,\, V \cup V',\, \preceq) \\ &\quad \text{onde } (E,\, V) = \text{consteq(c, V)} \\ \text{consteq}(+\text{f} : \tau_1 \to \tau_2 \to \tau_3,\, V) &= (\{[\![\tau_3]\!] \equiv ([\![\tau_2]\!] \ominus \emptyset) \oplus \{(f,\, \tau_1)\},\, V) \\ \text{consteq}(-\text{f} : \tau,\, V) &= (\{r_{\alpha_1} \equiv ([\![\tau]\!] \ominus \emptyset) \oplus \{(f,\, \alpha_2)\},\, V \cup \{\alpha_1, \alpha_2\}) \\ &\quad \text{onde} \quad \alpha_1 = \min_{\preceq}(\mathbf{V} - V) \text{ e} \\ &\qquad\quad\;\; \alpha_2 = \min_{\preceq}(\mathbf{V} - V - \{\alpha_1\}) \end{aligned}$$

$$\text{consteq}(c,\, V) = (\emptyset,\, V) \text{ se } c \text{ não for uma restrição de registro}$$

O objetivo desta teoria é prover uma função que compute a substituição principal de um conjunto de restrições de registro, caso pelo menos uma solução exista, ou que indique a insolubilidade do problema. Esta função é definida formalmente abaixo:

Definição 6.142.

$$\mathit{princRecSubst}(\kappa,\, \Gamma,\, \preceq) = \begin{cases} \bot & \text{se probinstance}(\kappa,\, \Gamma,\, \preceq) \xrightarrow{!} \bot \text{ ou se} \\ & \text{probinstance}(\kappa,\, \Gamma,\, \preceq) \xrightarrow{!} (E,\, S,\, V) \text{ e} \\ & \quad E \text{ é recursivo} \\ S & \text{probinstance}(\kappa,\, \Gamma,\, \preceq) \xrightarrow{!} (E,\, S,\, V) \text{ e} \\ & \quad E \text{ não é recursivo} \end{cases}$$

onde $p \xrightarrow{!} p'$ é redutítvel a forma $\mathcal{W}$-normal p'.

6.12 Exemplo de redução

Diversos exemplos do uso das regras de redução são apresentados no apêndice. Repetiremos aqui o exemplo A.14:

Seja κ o conjunto de restrições definido como:

$$\left\{ \begin{array}{l} \text{f} :: \alpha_1 \to \alpha_2 \to \alpha_3 \to \alpha_4, \quad +\text{f} :: \alpha_5 \to \alpha_1 \to \alpha_3, \\ +\text{f} :: \text{Int} \to \alpha_2 \to \alpha_3 \end{array} \right\}$$

Tomando $\preceq$ como a relação tal que $\alpha_i \preceq \alpha_k$ se e somente se $i \preceq k$, e considerando o contexto vazio, temos:

$$\text{probinstance}(\kappa,\, \Gamma,\, \preceq) = \left(\left\{ \begin{array}{lcl} r_{\alpha_3} & \equiv & (r_{\alpha_1} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \alpha_5) \ \right\} \\ r_{\alpha_3} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \mathrm{Int}) \ \right\} \end{array} \right\}, \mathrm{id}, \emptyset \right)$$

Esta redução procede como descrito abaixo, com a notação $p \xrightarrow{n} p'$ indicando a redução através da regra 1 e da regra de número n (com $n \geq 2$). Se a regra se aplica a um subconjunto próprio das equações de uma instância de problema, a regra 1 é usada para decompor o sistema em duas partes, uma das quais é o subconjunto em questão. As equações reduzidas pela regra n são sublinhadas, enquanto que as equações resultantes da redução pela regra n são indicadas por um triângulo ($\triangleright$).

$$\left(\left\{ \begin{array}{lcl} \underline{r_{\alpha_3}} & \equiv & \underline{(r_{\alpha_1} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \alpha_5) \ \right\}} \\ \underline{r_{\alpha_3}} & \equiv & \underline{(r_{\alpha_2} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \mathrm{Int}) \ \right\}} \end{array} \right\}, \mathrm{id}, \emptyset \right) \xrightarrow{8}$$

$$\left(\left\{ \begin{array}{lcl} \triangleright\, r_{\alpha_3} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \mathrm{Int}) \ \right\} \\ \triangleright\, \underline{r_{\alpha_2}} & \equiv & \underline{(r_{\alpha_1} \ominus \emptyset) \oplus \emptyset} \end{array} \right\}, [\alpha_5 \mapsto \mathrm{Int}], \emptyset \right) \xrightarrow{5}$$

$$\left(\left\{ r_{\alpha_3} \equiv (r_{\alpha_1} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \mathrm{Int}) \ \right\} \right\}, [\alpha_2 \mapsto \alpha_1, \alpha_5 \mapsto \mathrm{Int}], \{\alpha_1\} \right)$$

É fácil ver que nenhuma das variáveis é auto-recursiva e, portanto, a substituição principal é:

$$S = [\alpha_2 \mapsto \alpha_1, \alpha_5 \mapsto \mathrm{Int}]$$

e o conjunto de restrições especializado é:

$$S(\kappa) = \left\{ \ \mathrm{f} :: \alpha_1 \to \alpha_1 \to \alpha_3 \to \alpha_4, \quad +\mathrm{f} :: \mathrm{Int} \to \alpha_1 \to \alpha_3 \ \right\}$$

Capítulo 7

Sintaxe de usuário para polimorfismo de registro

Os capítulos anteriores descreveram um sistema de registros baseado no Sistema CT, que codifica o polimorfismo de registro usando dois novos tipos de restrições. No entanto, as restrições de registro não são uma forma amigável de representar polimorfismo de registro. A razão é que, para que o usuário entenda a informação codificada na restrições, o mesmo precisa fazer uma série de deduções, especialmente quando há um conjunto grande de restrições de registro.

Este capítulo descreve uma forma diferente de codificar polimorfismo de registro, que não faz uso de alterações na sintaxe de restrições, mas sim na sintaxe de tipos simples. Esta sintaxe, além de ser mais amigável, não usa restrições de registro, mas sim algumas construções sintáticas de tipo simples adicionais.

No entanto, veremos que esta sintaxe nova é apenas *açúcar sintático*, podendo ser convertida de e para tipos com restrição de registro. Assim, podemos usá-la externamente, nas anotações de tipo do programador, ou nas mensagens de erro, enquanto usamos as restrições de registro internamente no compilador e inferidor de tipos.

7.1 Sintaxe amigável

Começamos fazendo uma redefinição da sintaxe de tipos:

Variáveis de tipo	$\alpha, \beta \in \mathbf{V}$		
Construtores de tipo	$C \in \mathbf{C}$	$::= \mathrm{C}_n$	$n \geq 0$
Expressões de tipo simples	$\tau \in \mathbf{T}$	$::= \alpha \mid C \mid \tau_1\, \tau_2 \mid$	
		$\{\} \mid \{\mathrm{f}_1 : \tau_1, \ldots, \mathrm{f}_n : \tau_n\} \mid$	
		$\alpha \prec \{r\} \mid \alpha + \{z\}$	
Linhas de subtipagem	r	$::= \mathrm{f} : \tau,\, r \mid -\mathrm{f},\, r \mid \ldots$	
Linhas de modificação	z	$::= \sqcup \mid \mathrm{f} : \tau,\, z \mid -\mathrm{f},\, z \mid +\mathrm{f} : \tau,\, z$	
Restrições	$\kappa \in \kappa$	$::= x : \tau$	

onde $\sqcup$ é a *string* vazia.

A modificação consiste na remoção das restrições de registro e no acréscimo de duas construções às expressões de tipo simples: a construção de subtipagem de registro e a construção de modificação de registro.

Subtipagem de registro

Uma expressão de linha de subtipagem (r) consiste em um conjunto de especificações positivas e negativas de campos de registro, no máximo uma para cada nome de campo. Especificações positivas tem a forma $\mathrm{f} : \tau$, enquanto que especificações negativas tem a forma $-\mathrm{f}$. Uma expressão com a forma $\alpha \prec \{r\}$ especifica que α só pode ser instanciada para um tipo registro e também informações positivas e negativas sobre os campos que o tipo registro pode conter. Por exemplo, a expressão:

$$\alpha \prec \{\mathrm{f}_1 : \tau_1,\ -\mathrm{f}_2,\ \mathrm{f}_3 : \tau_3,\ \ldots\} \rightarrow (\tau_1, \tau_2)$$

representa o tipo de uma função cujo argumento só pode ser um registro que deve ter pelo menos os campos f_1, com tipo τ_1, e f_3, com tipo τ_3, mas que não pode ter um campo com nome f_2.

Usando esta sintaxe, os tipos das funções de projeção se tornam:

$$\#\,\mathrm{f} \ :\ \alpha \prec \{\mathrm{f} : \tau,\ \ldots\} \rightarrow \tau$$

Modificação de registro

Uma expressão de linha de modificação de registro (z) consiste em uma sequência de três tipos de especificação: especificação de atualização de campo, que tem a forma $\mathrm{f} : \tau$; especificação de remoção de campo, que tem a forma $-\mathrm{f}$; e especificação de adição de campo, que tem a forma $+\mathrm{f} : \tau$.

Uma expressão $\alpha + \{z\}$ representa tipos registro que devem ter os mesmos campos que o tipo registro para o qual α for instanciado, com a exceção das modificações listadas na expressão z.

Para que a expressão faça sentido, a variável α deve ocorrer ao menos uma vez no tipo em outra posição que não à esquerda do símbolo + em uma expressão de modificação de registro. Como um exemplo de expressão de modificação de registro, considere:

$$\alpha \to \alpha + \{\mathrm{f}_1 : \tau_1,\ +\mathrm{f}_2 : \tau_2,\ -\mathrm{f}_3\}$$

que representa o tipo de uma função que espera como parâmetro um registro e que retorna um registro com o tipo do campo f_1 alterado para τ_1, com um novo campo f_2 com tipo τ_2, e sem o campo f_3. Implicitamente, temos que α só pode ser instanciada para tipos registro que possuem os campos f_1 e f_3 mas não o campo f_2.

Usando esta sintaxe, os tipos das funções de remoção e adição de campo se tornam:

$$\begin{aligned} \#\text{-f} &: \alpha \to \alpha + \{-\mathrm{f}\} \\ \#+\mathrm{f} &: \tau \to \alpha \to \alpha + \{+\mathrm{f} : \tau\} \end{aligned}$$

Ou, usando também a sintaxe de subtipagem, temos os seguintes tipos equivalentes, mas com informações acerca de α apresentadas explicitamente:

$$\begin{aligned} \#\text{-f} &: \alpha \prec \{\mathrm{f} : \beta,\ \ldots\} \to \alpha + \{-\mathrm{f}\} \\ \#+\mathrm{f} &: \tau \to \alpha \prec \{-\mathrm{f},\ \ldots\} \to \alpha + \{+\mathrm{f} : \tau\} \end{aligned}$$

Sinônimos de tipos registro

Usando sinônimos de tipos, o polimorfismo de registro se torna bem expressivo. A figura 7.1 mostra um exemplo que define alguns tipos registro e operações que poderiam pertencer a uma biblioteca de componentes de interface de usuário.

A primeira novidade está na anotação para a função `button`:

```
button :: String → IO () → {Drawable,Dynamic}
```

A expressão `{Drawable,Dynamic}` é um sinônimo para `{width :: Int, height :: Int, draw :: Canvas → IO (), action :: IO () }`.

A anotação para a função `position` especifica que ela espera como segundo parâmetro um registro sem os campos presentes no sinônimo de tipos `Position`, retornando o mesmo registro com a adição dos campos de `Position`:

```
position :: Int → Int → a → a+{+Position}
```

A anotação de tipo para as funções `move` e `scale` especificam, respectivamente, que elas esperam como terceiro parâmetro um registro com os campos especificados nos sinônimos de tipo `Position` e `Drawable`, respectivamente.

```
type Position = { x :: Int, y :: Int }
type Drawable = {
     width :: Int,
     height :: Int,
     draw :: Canvas → IO () }
type Dynamic = { action :: IO () }
button :: String → IO () → {Drawable,Dynamic}
button msg action = {
   width = calcWidth msg,
   height = calcHeight msg,
   draw = λcanvas→...,
   action = action }
position :: Int → Int → a → a+{+Position}
position x y obj = { obj | +x ← x, +y ← y }
move :: Int → Int → a<{Position, ...} → a
move dx dy obj@{x, y, ...} = { obj | x ← x + dx, y ← y + dy}
scale :: Int → Int → a<{Drawable, ...} → a
scale sw sw obj@{width,height, ...} =
      { obj | width ← width * sw, height ← height * sh }
```

Figura 7.1: Sintaxe de usuário com sinônimos

Omissão de variáveis de tipo

Se a variável de tipo α aparece do lado esquerdo de uma expressão de subtipagem $\alpha \prec \{r\}$, mas não aparece em nenhuma outra parte do tipo, então a sub-expressão $\alpha \prec$ pode ser omitida. A expressão $\{r\}$ (que termina sempre com dois pontos para distinguí-la da expressão de tipo registro) se torna então um açúcar sintático para a expressão $\alpha' \prec \{r\}$, onde α' é uma variável nova.

Desta forma, o tipo das funções de projeção se tornam:

$$\#\,\mathrm{f} : \{\mathrm{f} : \tau, ...\} \rightarrow \tau$$

A próxima seção descreve como a sintaxe amigável pode ser traduzida de e para a sintaxe interna, que faz uso de restrições de registro.

7.2 Traduzindo da sintaxe de usuário para a sintaxe interna

A tradução da sintaxe de usuário para a sintaxe interna é simples. A figura 7.2 mostra um algoritmo de tradução em forma de regras de reescrita:

Por exemplo, o tipo:

$$
\begin{array}{rcl}
\forall\bar{\alpha}.\kappa.\tau & \gg_{U\to I} & \forall\bar{\alpha}\cup\bar{\beta}.\kappa_1\cup\kappa_2.\tau' \\
 & & \text{onde } \kappa \gg_{U\to I} \kappa_1 \\
 & & \text{ e } \tau \gg_{U\to I} \kappa_2.\tau' \\
 & & \text{ e } \bar{\beta} = (\mathrm{tv}(\kappa_1)\cup\mathrm{tv}(\kappa_2.\tau')) - \mathrm{tv}(\forall\bar{\alpha}.\kappa.\tau) \\
\alpha & \gg_{U\to I} & \kappa.\alpha \\
\tau_1\,\tau_2 & \gg_{U\to I} & \kappa_1\cup\kappa_2.\tau_1'\,\tau_2' \\
 & & \text{onde } \tau_1 \gg_{U\to I} \kappa_1.\tau_1 \\
 & \gg_{U\to I} & \text{ e } \tau_2 \gg_{U\to I} \kappa_2.\tau_2 \\
\{\mathrm{f}_1:\tau_1,\ldots,\mathrm{f}_n:\tau_n\} & \gg_{U\to I} & \bigcup_i \kappa_i.\{\mathrm{f}:\tau_1',\cdots,\mathrm{f}:\tau_n'\} \\
 & & \text{onde } \tau_i \gg_{U\to I} \kappa_i.\tau_i' \\
\alpha\prec\{\mathrm{f}:\tau,r\} & \gg_{U\to I} & \{+\mathrm{f}\ :\ \tau'\to\beta\to\alpha\}\cup\kappa\cup\kappa'\,.\,\tau_\alpha \\
 & & \text{onde } \tau \gg_{U\to I} \kappa.\tau', \\
 & & \qquad \alpha\prec\{r\} \gg_{U\to I} \kappa'\,.\,\tau_\alpha \\
 & & \text{ e } \beta \text{ é uma variável nova} \\
\alpha\prec\{-\mathrm{f},r\} & \gg_{U\to I} & \{-\mathrm{f}\ :\ \alpha\}\cup\kappa\,.\,\tau \\
 & & \text{onde } \alpha\prec\{r\} \gg_{U\to I} \kappa.\tau \\
\alpha\prec\{\ldots\} & \gg_{U\to I} & \emptyset.\alpha \\
\alpha+\{\mathrm{f}:\tau,z\} & \gg_{U\to I} & \{+\mathrm{f}\ :\ \beta_1\to\beta_2\to\alpha\}\cup\kappa.\tau' \\
 & & \text{onde } \beta_2+\{+\mathrm{f}:\tau,z\} \gg_{U\to I} \kappa\,.\,\tau' \\
 & & \text{ e } \beta_1,\beta_2 \text{ são variáveis novas} \\
\alpha+\{+\mathrm{f}:\tau,z\} & \gg_{U\to I} & \{+\mathrm{f}\ :\ \tau\to\alpha\to\beta\}\cup\kappa.\tau' \\
 & & \text{onde } \beta+\{z\} \gg_{U\to I} \kappa\,.\,\tau' \\
 & & \text{ e } \beta \text{ é uma variável nova} \\
\alpha+\{-\mathrm{f},z\} & \gg_{U\to I} & \{+\mathrm{f}\ :\ \beta_1\to\beta_2\to\alpha\}\cup\kappa.\tau' \\
 & & \text{onde } \beta_2+\{z\} \gg_{U\to I} \kappa\,.\,\tau' \\
 & & \text{ e } \beta_1,\beta_2 \text{ são variáveis novas} \\
\alpha+\{\} & \gg_{U\to I} & \emptyset.\alpha \\
\{x:\tau\}\cup\kappa & \gg_{U\to I} & \{x:\tau'\}\cup\kappa_1\cup\kappa_2 \\
 & & \text{onde } \tau \gg_{U\to I} \kappa_1.\tau' \\
 & & \text{ e } \kappa \gg_{U\to I} \kappa_2 \\
\emptyset & \gg_{U\to I} & \emptyset
\end{array}
$$

Figura 7.2: Tradução de sintaxe de usuário para sintaxe interna.

$$\forall\alpha.\emptyset.\alpha \prec \{\mathrm{f}_1 : \tau_1,\ \mathrm{f}_2 : \tau_2,\ -\mathrm{f}_3,\ \mathrm{f}_4 : \tau_4,\ ...\} \to \alpha + \{\mathrm{f}_2 : \tau_2',\ -\mathrm{f}_4,\ +\mathrm{f}_5 : \tau_5\}$$

seria reescrito como $\forall\bar{\alpha}.\kappa.\alpha \to \beta$, onde:

$$\begin{aligned}
\kappa &= \{+\mathrm{f}_1 : \tau_1 \to \alpha_1 \to \alpha,\ +\mathrm{f}_2 : \tau_2 \to \alpha_2 \to \alpha,\\
&\quad -\mathrm{f}_3 : \alpha,\ +\mathrm{f}_4 : \tau_4 \to \alpha_3 \to \alpha,\\
&\quad +\mathrm{f}_2 : \alpha_4 \to \alpha_5 \to \alpha,\ +\mathrm{f}_2 : \tau_2' \to \alpha_5 \to \alpha_6,\\
&\quad +\mathrm{f}_4 : \alpha_7 \to \alpha_8 \to \alpha_6, \quad +\mathrm{f}_5 : \tau_5 \to \alpha_8 \to \beta\}\\
\bar{\alpha} &= \{\alpha, \alpha_1, \ldots, \alpha_8, \beta\}
\end{aligned}$$

Pode se ver claramente neste exemplo que a sintaxe de usuário é mais breve e intuitiva que a sintaxe interna.

7.3 Traduzindo da sintaxe interna para a sintaxe de usuário

A tradução da sintaxe interna para a sintaxe de usuário é baseada no sistema de redução $\mathcal{W}$, definido no capítulo 6.

Primeiramente, precisamos notar que a tradução só é bem definida para tipos restringidos cujos conjuntos de restrições de registros são mínimos. Em outras palavras, é preciso computar a substituição principal do conjunto de restrições de registro e especializá-lo antes de fazer a tradução, que é feita segundo a definição:

Definição 7.1. Dado um contexto Γ e uma ordem total $\preceq$ sobre variáveis de tipo, as restrições de registro são consideradas mínimas se

$$\mathrm{probinstance}(\kappa,\ \Gamma,\ \preceq) \xrightarrow[\mathcal{W}]{!} (E, \mathrm{id}, V)$$

Onde a função probinstance e o sistema de redução $\underset{\mathcal{W}}{\longrightarrow}$ são definidos no capítulo 6.

Deve ser notado que se o conjunto de restrições de registro do tipo restringido $\kappa.\tau$ não for mínimo, mas se a instância de problema relacionada tiver uma forma normal, i.e.

$$\mathrm{probinstance}(\kappa,\ \Gamma,\ \preceq) \xrightarrow[\mathcal{W}]{!} (E, \mathrm{S}, V)$$

então $S(\kappa.\tau)$ é um tipo restringido mínimo (com respeito às restrições de registro) equivalente a $\kappa.\tau$ no contexto Γ.

A consequência desta exigência é que não é possível usar a sintaxe de usuário para expressar tipos restringidos que ainda não foram simplificados pelo processo de cálculo e aplicação da menor generalização comum das soluções de suas restrições de registro. Outras alternativas devem ser buscadas, nestes casos, para não expor a sintaxe interna ao usuário.

A partir da forma normal correspondente ao conjunto de restrições de registro do tipo em questão, definimos a representação, através de sintaxe de usuário, de cada variável α que aparece no tipo, e que está presente nas restrições de registro.

Esta representação é definida pela função $\mathrm{rep}(\alpha,\, E)$:

$$\mathrm{rep}(\alpha,\, E) = \begin{cases} \alpha \prec \{f_1 : \tau_1, \ldots, f_n : \tau_n, -f'_1, \ldots, -f'_j\} & \text{se } \alpha \in \mathrm{rvars}(E) \\ & \text{onde } (f_i, \tau_i) \in \mathrm{posR}(\alpha,\, E) \\ & \text{e } f'_i \in \mathrm{negR}(\alpha,\, \mathrm{E}) \\ \beta + \{f_1^{mod} : \tau_1^{mod}, \ldots, f_n^{mod} : \tau_n^{mod}, & \text{se } \alpha \in \mathrm{lvars}(E) \\ \quad +f_1^{new} : \tau_1^{new}, \ldots, \mathrm{f}_m^{neq} : \tau_m^{new}, & \text{onde } \beta = \mathrm{rvar}(\alpha,\, E) \\ \quad -f_1^{rem}, \ldots, -f_k^{rem}\} & \text{e } (f_i^{mod}, \tau_i^{mod}) \in \mathrm{modL}(\alpha,\, E) \\ & \text{e } (f_i^{new},\, \tau_i^{new}) \in \mathrm{newL}(\alpha,\, E) \\ & \text{e } f_i^{rem} \in \mathrm{remL}(\alpha,\, E) \\ \alpha & \text{caso contrário} \end{cases}$$

$$\mathrm{rvars}(E) = \{\alpha \mid r_\beta \equiv (r_\alpha \ominus R^-) \oplus R^+ \in E\}$$

$$\mathrm{lvars}(E) = \{\alpha \mid r_\alpha \equiv (r_\beta \ominus R^-) \oplus R^+ \in E \text{ e } \alpha \notin \mathrm{rvars}(E)\}$$

$$\mathrm{posR}(\alpha,\, E) = \bigcup\{R^- \mid r_\beta \equiv (r_\alpha \ominus R^-) \oplus R^+ \in E\}$$

$$\mathrm{negR}(\alpha,\, E) = \bigcup\{\mathrm{fnames}(R^+) \mid r_\beta \equiv (r_\alpha \ominus R^-) \oplus R^+ \in E\}$$

$$\mathrm{rvar}(\alpha,\, E) = \beta \text{ onde } r_\alpha \equiv (r_\beta \ominus R^-) \oplus R^+ \in E$$

$$\mathrm{modL}(\alpha,\, E) = \{(f,\, \tau) \mid r_\alpha \equiv (r_\beta \ominus R^-) \oplus R^+ \text{ e } (f,\, \tau) \in R^+ \text{ e } f \in \mathrm{fnames}(R^-)\}$$

$$\mathrm{newL}(\alpha,\, E) = \{(f,\, \tau) \mid r_\alpha \equiv (r_\beta \ominus R^-) \oplus R^+ \text{ e } (f,\, \tau) \in R^+ \text{ e } f \notin \mathrm{fnames}(R^-)\}$$

$$\mathrm{remL}(\alpha,\, E) = \{f \mid r_\alpha \equiv (r_\beta \ominus R^-) \oplus R^+ \text{ e } f \in \mathrm{fnames}(R^-) - \mathrm{fnames}(R^+)\}$$

A tradução de um tipo restringido $\kappa.\tau$ em um contexto Γ usando uma ordem total $\preceq$ sobre variáveis de tipo é definida pela função $\mathrm{touser}(\kappa.\tau,\, \Gamma,\, \preceq)$:

Definição 7.2.

$$\text{touser}(\kappa.\tau,\, \Gamma,\, \preceq) = \text{mapCT}(\kappa.\tau,\, E) \text{ se probinstance}(\kappa,\, \Gamma,\, \preceq) \xrightarrow[\mathcal{W}]{!} (E,\, S,\, V)$$

$$\text{mapCT}(\kappa.\tau,\, E) = \kappa'.\tau' \text{ onde } (\tau',\, \mathrm{V}) = \text{mapT}(\kappa, \emptyset) \text{ e } \kappa' = \text{mapC}(\kappa, \mathrm{V})$$

$$\begin{aligned}
\text{mapT}(\alpha,\, V) &= \begin{cases} (\text{rep}(\alpha,\, E),\, V \cup \{\alpha\}) & \text{se } \alpha \notin V \\ (\alpha,\, V) & \text{caso contrário} \end{cases} \\
\text{mapT}(C,\, V) &= (C,\, V) \\
\text{mapT}(\tau_1\, \tau_2) &= (\tau_1'\, \tau_2',\, V'') \text{ onde } \begin{array}{l} (\tau_1',\, V') = \text{mapT}(\tau_1,\, V) \text{ e} \\ (\tau_2',\, V'') = \text{mapT}(\tau_2,\, V') \end{array} \\
\text{mapT}(\{f_1 : \tau_1, \ldots, f_n : \tau_n\}, V) &= (\{f_1 : \tau_1', \ldots, f_n : \tau_n'\},\, V_n) \\
&\qquad \text{onde } \begin{array}{l} (\tau_i',\, V_i) = \text{mapT}(\tau_i,\, V_{i-1}) \text{ e} \\ V_0 = V \end{array}
\end{aligned}$$

$$\begin{aligned}
\text{mapC}(\{x : \tau\} \cup \kappa,\, V) &= \{x : \tau'\} \cup \kappa' \text{ onde } \begin{array}{l} (\tau',\, V') = \text{mapT}(\tau,\, V) \text{ e} \\ \kappa' = \text{mapC}(\kappa,\, V') \end{array} \\
\text{mapC}(\{+f : \tau\} \cup \kappa,\, V) &= \text{mapC}(\kappa,\, V) \\
\text{mapC}(\{-f : \tau\} \cup \kappa,\, V) &= \text{mapC}(\kappa,\, V)
\end{aligned}$$

7.4 Exemplo

O exemplo A.15, descrito no apêndice, mostra um tipo restringido juntamente com a instância de problema correspondente, bem como a forma normal desta última, à partir da qual o tipo com sintaxe de usuário é construído. Repetiremos aqui algumas partes do exemplo, como ilustração.

Considere a expressão:

$$\text{let } f(x) = \{x \mid \mathrm{f} \leftarrow \text{"}\texttt{string}\text{"}, +\mathrm{g} \leftarrow \mathrm{True}, -\mathrm{h}\}$$

que pode ser traduzida para:

$$\text{let } f(x) = \#-\mathrm{h}(\#+\mathrm{g}\ \mathrm{True}(\#+\mathrm{f}\ \text{"}\texttt{string}\text{"}(\#-\mathrm{f}\ x)))$$

O algoritmo de inferência de tipos infere, primeiramente, o seguinte tipo restringido para a função f:

$$\left\{ \begin{array}{ll} +\mathrm{f} :: \alpha_1 \to \alpha_2 \to \alpha_3, & +\mathrm{f} :: \mathrm{String} \to \alpha_2 \to \alpha_4, \\ +\mathrm{g} :: \mathrm{Bool} \to \alpha_4 \to \alpha_5, & +\mathrm{h} :: \alpha_6 \to \alpha_7 \to \alpha_5 \end{array} \right\} . \alpha_3 \to \alpha_3 \to \alpha_7$$

A instância de problema correspondente é:

$$\left(\left\{ \begin{array}{lcl} r_{\alpha_3} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \{\ (\mathrm{f}, \alpha_1)\ \} \\ r_{\alpha_4} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \{\ (\mathrm{f}, \mathrm{String})\ \} \\ r_{\alpha_5} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \{\ (\mathrm{g}, \mathrm{Bool})\ \} \\ r_{\alpha_5} & \equiv & (r_{\alpha_7} \ominus \emptyset) \oplus \{\ (\mathrm{h}, \alpha_6)\ \} \end{array} \right\}, \mathrm{id}, \emptyset \right)$$

E a forma normal é:

$$\left(\left\{ \begin{array}{rcl} r_{\alpha_2} & \equiv & \big(r_{\alpha_3} \ominus \{\ (\mathrm{f}, \alpha_1)\ \}\big) \oplus \emptyset \\ r_{\alpha_5} & \equiv & \big(r_{\alpha_3} \ominus \{\ (\mathrm{f}, \alpha_1)\ \}\big) \oplus \{\ (\mathrm{f}, \mathrm{String}),\ \ (\mathrm{g}, \mathrm{Bool})\ \} \\ \rhd\, r_{\alpha_4} & \equiv & \big(r_{\alpha_3} \ominus \{\ (\mathrm{f}, \alpha_1),\ \ (\mathrm{h}, \alpha_6)\ \}\big) \oplus \{\ (\mathrm{f}, \mathrm{String}),\ \ (\mathrm{h}, \alpha_6)\ \} \\ \rhd\, r_{\alpha_7} & \equiv & \big(r_{\alpha_3} \ominus \{\ (\mathrm{f}, \alpha_1),\ \ (\mathrm{h}, \alpha_6)\ \}\big) \oplus \{\ (\mathrm{f}, \mathrm{String}),\ \ (\mathrm{g}, \mathrm{Bool})\ \} \end{array} \right\}, \mathrm{id}, \emptyset \right)$$

à partir da qual podemos derivar o tipo correspondente usando sintaxe de usuário:

$$\alpha_3 \prec \{\mathrm{f} : \alpha_1,\ -\mathrm{g},\ \mathrm{h} : \alpha_6,\ \ldots\} \to \alpha_3 \to \alpha_3 + \{\mathrm{f} : \mathrm{String},\ +\mathrm{g} : \mathrm{Bool},\ -\mathrm{h}\}$$

Capítulo 8

Implementação

Um protótipo do algoritmo de cálculo da substituição principal de conjuntos de restrições de registro (capítulo 6) e da tradução da sintaxe interna para a sintaxe de usuário (capítulo 7) foi feito em Haskell.

Dois testes foram implementados. O primeiro lê uma lista de conjuntos de restrições de um arquivo especificado pelo usuário, com um conjunto por linha, e gera um arquivo fonte LaTeXpara cada um dos conjuntos de restrição, mostrando a derivação da substituição principal das restrições de registro, ou da prova de que nenhuma solução existe.

O segundo teste lê uma lista de tipos restringidos de um arquivo especificado pelo usuário e gera arquivos LaTeXcom a derivação da substituição principal, ou prova de insolubilidade, e o tipo restringido resultante usando sintaxe de usuário, como definido no capítulo 7.

8.1 Arquitetura do protótipo

O código do protótipo foi organizado em 7 agrupamentos de módulos de Haskell:

- Estrutura de dados que representam tipos restringidos, conjuntos de campos, termos de conjuntos de campos, equações de conjuntos de campo e instâncias de problemas, bem como de funções relacionadas, foram definidas nos módulos:
  - SystemW.Data.Types
  - SystemW.Data.FieldSet
  - SystemW.Data.FieldSetTerms
  - SystemW.Data.FieldSetEquation
  - SystemW.Data.EquationSystem
  - SystemW.Data.ProblemInstance
- O sistema de redução e funções relacionadas a sistemas de equação $\preceq$-normais foram definidos em:

  - SystemW.Algorithm.ReductionSystem
  - SystemW.Algorithm.NormalEquationSystem

- O algoritmo de tradução da sintaxe interna para a sintaxe de usuário foi definido no módulo:

  - SystemW.UserSyn.UserSyntax

- O parser para a sintaxe interna foi definido em:

  - SystemW.Parser.Lexer
  - SystemW.Parser.Parser

- A saída em LaTeXdos tipos de dados, mensagens de erro e derivações geradas pelos algoritmos foi definida nos módulos:

  - SystemW.Latex.Latex
  - SystemW.Latex.Types
  - SystemW.Latex.FieldSet
  - SystemW.Latex.FieldSetTerms
  - SystemW.Latex.FieldSetEquation
  - SystemW.Latex.EquationSystem
  - SystemW.Latex.ProblemInstance
  - SystemW.Latex.ReductionSystem
  - SystemW.Latex.UserSyntax

- A saída em forma de string das estruturas de dados e mensagens de erro foi definida em:

  - SystemW.Show.EquationSystem
  - SystemW.Show.FieldSetEquation
  - SystemW.Show.FieldSet
  - SystemW.Show.FieldSetTerms
  - SystemW.Show.ProblemInstance
  - SystemW.Show.ReductionSystem
  - SystemW.Show.Types

- Módulos de teste:

  - Main
  - ProbInstanceTest
  - UserSyntaxTest

8.2 Disponibilidade

A última versão do código fonte deste protótipo deve estar disponível em [2] ou por requisição ao autor.

Capítulo 9

Conclusão

O objetivo proposto para este trabalho foi o de projetar um sistema de registros para o Sistema CT e avaliar como o sistema de restrições do Sistema CT pode facilitar este tipo de projeto. O trabalho procedeu-se através dos seguintes passos:

- Primeiramente, o autor efetuou uma revisão da linguagem Haskell e de seu sistema de registros, descrevendo suas características e limitações.
- Também foi feita uma revisão do sistema de tipos do Sistema CT, na qual foram apresentadas algumas contribuições, como a reformulação da sintaxe de tipos, da função lcg e das regras de satisfazibilidade.
- Seguiu-se uma argumentação sobre as características desejáveis de um sistema de registro, bem como a sintaxe e regras de tipagem para um sistema de registros para o Sistema CT. As principais características do sistema proposto são:
  - Registros *leves*: o uso de um tipo registro não precisa ser precedido de uma declaração do mesmo.
  - Operações polimórficas de registro: projeção, adição, remoção e atualização de campos.
  - Codificação de informações sobre polimorfismo de registro usando restrições de registro.
- A teoria de equações de conjuntos de campos, que é a maior contribuição deste trabalho, foi proposta para formalizar o problema da substituição principal para restrições de registro, e para permitir a formulação de um algoritmo para computar esta substituição principal através de um sistema de redução. Uma prova da terminação do sistema de redução, bem como um esqueleto da prova da consistência das regras de redução também foram apresentados neste trabalho.
- Uma sintaxe especial para tipos com registros polimórficos foi proposta, com o objetivo de prover uma codificação mais intuitiva de tipos restringidos com restrições

de registro. Foi mostrado uma mapeamento entre a sintaxe que usa restrições de registro e a sintaxe amigável ao usuário.

- Por último, foi desenvolvido um protótipo do sistema de redução e do mapeamento entre as sintaxes.

Apesar de ser baseado no sistema de sobrecarga do Sistema CT, o sistema de registros proposto pode ser também formulado, com poucas modificações, usando o sistema de classes de Haskell.

O autor espera que este trabalho possa contribuir à pesquisa corrente sobre sistemas de registro em Haskell e sobre sistemas de tipos em geral.

Trabalhos futuros

Este documento descreveu um sistema de registro para o Sistema CT sob a forma de regras de tipagem, regras de satisfazibilidade e de um algoritmo de cálculo de substituição principal. No entanto, um assunto que deve ser tratado antes que o sistema possa ser efetivamente implementado é a semântica operacional do sistema, que deve levar em conta questões sobre a eficiência da manipulação dos registros em tempo de execução.

Nenhum compilador foi ainda implementado para o Sistema CT até o momento da publicação deste trabalho. Portanto, a implementação de um compilador com suporte ao sistema de registros proposto seria um trabalho interessante.

Outra possibilidade é a de integrar o sistema proposto a um compilador existente de Haskell, mostrando as mudanças necessárias para que a linguagem Haskell possa usar o sistema de registro proposto.

Outros acréscimos que poderiam ser feitos ao sistema são:

- Concatenação de registros: o sistema de tipos proposto não consegue expressar um tipo para a operação de concatenação de registros.

- Registros de *rank* superior: registros com campos polimórficos podem ser usados para implementar módulos de primeira classe.

Referências Bibliográficas

[1] The haskell mailing lists. `http://www.haskell.org/haskellwiki/Mailing_lists`.

[2] System w prototype. `http://dcc.ufmg.br/~camarao/records/systemw/_darcs`.

[3] The ghc survey. `http://www.haskell.org/ghc/survey2005-summary.html`, 2005.

[4] Martín Abadi and Luca Cardelli. A semantics of object types. In *Ninth Annual IEEE Symposium on Logic in Comptuer Science, Paris, France*, pages 332–341, Los Alamitos, CA, 1994.

[5] H. A. Priestley B. A. Davey. *Introduction to Lattices and Order*. Cambridge Press, 2nd edition, April 2002.

[6] Carlos Camarão, Lucília Figueiredo, and Cristiano Vasconcellos. Constraint-set Satisfiability for Overloading. In *Proc. of ACM PPDP'04*, pages 67–77, 2004.

[7] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 2nd IEEE Symposium on Logic in Computer Science*, 1982. Corrigendum, Proc. 3rd IEEE Symposium on Logic in Computer Science, 1988, page 132.

[8] Luís Damas and Robin Milner. Principal type schemes for functional programs. In *Proc. of POPL'82*, pages 207–212, 1982.

[9] E. Eder. Properties of substitutions and unifications. *Journal of Symbolic Computation*, 1:31–46, 1985.

[10] B. Gaster and M. Jones. A polymorphic type system for extensible records and variants. Technical report, University of Nottingham, November 1996.

[11] Benedict R. Gaster and Mark P. Jones. A Polymorphic Type System for Extensible Records and Variants. Technical Report NOTTCS-TR-96-3, Languages and Programming Group, Department of Computer Science, Nottingham NG7 2RD, UK, November 1996.

[12] Fritz Henglein. Type inference with polymorphic recursion. *ACM TOPLAS*, 15(2):253–289, Apr 1993.

[13] K. Marriot J. L. Lassez, M. J. Maher. Unification revisited. 1988.

[14] S. Jahama and A.J. Kfoury. A General Theory of Semi-Unification. Technical Report bu-cs 93-018, Boston University, 1993.

[15] Trevor Jim. What are principal typings and what are they good for? In ACM, editor, *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 42–53, 1996.

[16] M. Jones and S. Jones. Lightweight extensible records for haskell, 1999.

[17] Mark P. Jones. Simplifying and improving qualified types. Technical Report YALEU/DCS/RR-1040, Dept. of Computer Science, Yale University, 1994.

[18] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, April 2003.

[19] Simon Peyton Jones and Greg Morrisett. A proposal for records in haskell. available at http://research.microsoft.com/ simonpj/Haskell/records.html, Feb 2003.

[20] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, April 1993.

[21] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 1(102):83–101, 1993.

[22] Oleg Kiselyov and Ralf Lämmel. Haskell's overlooked object system. 2005.

[23] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. Technical Report SEN-E0420, CWI, Amsterdam, aug 2004.

[24] Daan Leijen. First-class labels for extensible rows. Technical Report UU-CS-2004-51, Department of Computer Science, Universiteit Utrecht, dec 2004.

[25] Daan Leijen. Extensible records with scoped labels. In *Proceedings of the 2005 Symposium on Trends in Functional Programming (TFP'05)*, sep 2005.

[26] Daan Leijen. Morrow: a row-oriented programming language. `http://www.cs.uu.nl/~daan/morrow.html`, January 2005.

[27] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[28] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1997.

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

[30] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *Proceedings of the International Symposium on Programming*, pages 217–239, 1984. Volume 167 of LNCS, Springer-Verlag.

[31] Atsushi Ohori. A compilation method for ml-style polymorphic record calculi. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 154–165, New York, NY, USA, 1992. ACM Press.

[32] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, November 1995.

[33] Catuscia Palamidessi. Algebraic properties of idempotent substitutions. October 1996.

[34] Benjamin C. Pierce. *Basic category theory for computer scientists.* MIT Press, Cambridge, MA, USA, 1991.

[35] Geoffrey Smith. *Polymorphic Type Inference for Languages with Overloading and Subtyping.* PhD thesis, Cornell University, 1991.

[36] Cristiano Vasconcelos, Lucília Figueiredo, and Carlos Camarão. Practical Type Inference for Polymorphic Recursion: an Implementation in Haskell. *Journal of Universal Computer Science*, 9(8):873–890, 2003.

[37] Cristiano Vasconcelos, Lucília Figueiredo, and Carlos Camarão. Practical Type Inference for Polymorphic Recursion: an Implementation in Haskell. In *Proc. of SBLP'2003*, pages 166–179, 2003.

[38] Dennis Volpano and Geoffrey Smith. On the Complexity of ML Typability with Overloading. In *Proc. of the ACM Symposium on Functional Programming Computer Architecture.*, number 523 in LNCS, pages 15–28, 1991.

Apêndice

Apêndice A

Exemplos de redução do Sistema $\mathcal{W}$

Exemplo A.1. Conjunto de restrições:

$$\left\{\ +\mathrm{f} :: \mathrm{Int} \to \left\{\ \mathrm{h} : \mathrm{Int},\ \ \mathrm{k} : \mathrm{String}\ \right\} \to \alpha_1\ \right\}$$

Equações :

$$\left\{ r_{\alpha_1} \equiv \left(\left\{\ (\mathrm{h}, \mathrm{Int}),\ \ (\mathrm{k}, \mathrm{String})\ \right\} \ominus \emptyset\right) \oplus \left\{\ (\mathrm{f}, \mathrm{Int})\ \right\}\right\}$$

Redução:

$$\left(\left\{ r_{\alpha_1} \equiv \left(\left\{\ (\mathrm{h}, \mathrm{Int}),\ \ (\mathrm{k}, \mathrm{String})\ \right\} \ominus \emptyset\right) \oplus \left\{\ (\mathrm{f}, \mathrm{Int})\ \right\}\right\},\ \mathrm{id},\ \emptyset\right) \xrightarrow{2}$$

$$\left(\emptyset,\ \left[\alpha_1 \mapsto \left\{\ \mathrm{f} : \mathrm{Int},\ \ \mathrm{h} : \mathrm{Int},\ \ \mathrm{k} : \mathrm{String}\ \right\}\right],\ \emptyset\right)$$

O conjunto é satisfazível. A substituição principal é:

$$\left[\alpha_1 \mapsto \left\{\ \mathrm{f} : \mathrm{Int},\ \ \mathrm{h} : \mathrm{Int},\ \ \mathrm{k} : \mathrm{String}\ \right\}\right]$$

Exemplo A.2. Conjunto de restrições:

$$\left\{\ +\mathrm{f} :: \mathrm{Int} \to \left\{\ \mathrm{h} : \mathrm{Int},\ \ \mathrm{k} : \mathrm{String}\ \right\} \to \alpha_1,\ \ +\mathrm{g} :: \mathrm{Double} \to \alpha_2 \to \alpha_1\ \right\}$$

Equações :

$$\left\{\begin{array}{lcl} r_{\alpha_1} & \equiv & \left(\left\{\ (\mathrm{h}, \mathrm{Int}),\ \ (\mathrm{k}, \mathrm{String})\ \right\} \ominus \emptyset\right) \oplus \left\{\ (\mathrm{f}, \mathrm{Int})\ \right\} \\ r_{\alpha_1} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \left\{\ (\mathrm{g}, \mathrm{Double})\ \right\} \end{array}\right\}$$

Redução:

$$\left(\left\{\begin{array}{lcl} \underline{r_{\alpha_1}} & \equiv & \underline{\left(\left\{\ (\mathrm{h}, \mathrm{Int}),\ \ (\mathrm{k}, \mathrm{String})\ \right\} \ominus \emptyset\right) \oplus \left\{\ (\mathrm{f}, \mathrm{Int})\ \right\}} \\ r_{\alpha_1} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \left\{\ (\mathrm{g}, \mathrm{Double})\ \right\} \end{array}\right\},\ \mathrm{id},\ \emptyset\right) \xrightarrow{2}$$

$$\left(\left\{\left\{\ (\mathrm{f}, \mathrm{Int}),\ \ (\mathrm{h}, \mathrm{Int}),\ \ (\mathrm{k}, \mathrm{String})\ \right\} \equiv (r_{\alpha_2} \ominus \emptyset) \oplus \left\{\ (\mathrm{g}, \mathrm{Double})\ \right\}\right\},\ S,\ \emptyset\right) \xrightarrow{3} \bot$$

onde $\left[\alpha_1 \mapsto \left\{\ \mathrm{f} : \mathrm{Int},\ \ \mathrm{h} : \mathrm{Int},\ \ \mathrm{k} : \mathrm{String}\ \right\}\right]$

O conjunto não é satisfazível, pois $\{\mathrm{g}\} \not\subseteq \{\mathrm{f}, \mathrm{h}, \mathrm{k}\}$.

Exemplo A.3. Conjunto de restrições:

$$\left\{ \; +\mathrm{f} :: \mathrm{Int} \to \left\{ \; \mathrm{f} : \mathrm{Bool}, \quad \mathrm{h} : \mathrm{Int}, \quad \mathrm{k} : \mathrm{String} \; \right\} \to \alpha_1 \; \right\}$$

Equações :

$$\left\{ r_{\alpha_1} \equiv \left(\left\{ \; (\mathrm{f}, \mathrm{Bool}) \, , \quad (\mathrm{h}, \mathrm{Int}) \, , \quad (\mathrm{k}, \mathrm{String}) \; \right\} \ominus \emptyset \right) \oplus \left\{ \; (\mathrm{f}, \mathrm{Int}) \; \right\} \right\}$$

Redução:

$$\left(\left\{ r_{\alpha_1} \equiv \left(\left\{ \; (\mathrm{f}, \mathrm{Bool}) \, , \quad (\mathrm{h}, \mathrm{Int}) \, , \quad (\mathrm{k}, \mathrm{String}) \; \right\} \ominus \emptyset \right) \oplus \left\{ \; (\mathrm{f}, \mathrm{Int}) \; \right\} \right\}, \mathrm{id}, \emptyset \right) \xrightarrow{2} \bot$$

O conjunto não é satisfazível, pois: $\{\mathrm{f}, \mathrm{h}, \mathrm{k}\} \cap \{\mathrm{f}\} = \{\mathrm{f}\} \not\subseteq \emptyset$.

Exemplo A.4. Conjunto de restrições:

$$\left\{ \; +\mathrm{f} :: \mathrm{Int} \to \left\{ \; \mathrm{h} : \mathrm{Int}, \quad \mathrm{k} : \mathrm{String} \; \right\} \to \alpha_1, \quad +\mathrm{f} :: \mathrm{String} \to \alpha_2 \to \alpha_1 \; \right\}$$

Equações:

$$\left\{ \begin{array}{lcl} r_{\alpha_1} & \equiv & \left(\left\{ \; (\mathrm{h}, \mathrm{Int}) \, , \quad (\mathrm{k}, \mathrm{String}) \; \right\} \ominus \emptyset \right) \oplus \left\{ \; (\mathrm{f}, \mathrm{Int}) \; \right\} \\ r_{\alpha_1} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \left\{ \; (\mathrm{f}, \mathrm{String}) \; \right\} \end{array} \right\}$$

Redução:

$$\left(\left\{ \begin{array}{lcl} r_{\alpha_1} & \equiv & \left(\left\{ \; (\mathrm{h}, \mathrm{Int}) \, , \quad (\mathrm{k}, \mathrm{String}) \; \right\} \ominus \emptyset \right) \oplus \left\{ \; (\mathrm{f}, \mathrm{Int}) \; \right\} \\ r_{\alpha_1} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \left\{ \; (\mathrm{f}, \mathrm{String}) \; \right\} \end{array} \right\}, \mathrm{id}, \emptyset \right) \xrightarrow{2}$$

$$\left(\left\{ \left\{ \; (\mathrm{f}, \mathrm{Int}) \, , \quad (\mathrm{h}, \mathrm{Int}) \, , \quad (\mathrm{k}, \mathrm{String}) \; \right\} \equiv (r_{\alpha_2} \ominus \emptyset) \oplus \left\{ \; (\mathrm{f}, \mathrm{String}) \; \right\} \right\}, S, \emptyset \right) \xrightarrow{3} \bot$$
$$\text{onde } S = \left[\alpha_1 \mapsto \left\{ \; \mathrm{f} : \mathrm{Int}, \quad \mathrm{h} : \mathrm{Int}, \quad \mathrm{k} : \mathrm{String} \; \right\} \right]$$

O conjunto não é satisfazível, pois temos que o mgu de

$$\left\{ \left(\alpha_2, \left\{ \; \mathrm{h} : \mathrm{Int}, \quad \mathrm{k} : \mathrm{String} \; \right\} \right), (\mathrm{Int}, \mathrm{String}) \right\}$$

não é definido, pois os tipos Int e String não são unificáveis.

Exemplo A.5. Conjunto de restrições:

$$\left\{ \; +\mathrm{f} :: \mathrm{Int} \to \left\{ \; \mathrm{h} : \mathrm{Int}, \quad \mathrm{k} : \mathrm{String} \; \right\} \to \left\{ \; \mathrm{f} : \mathrm{Int}, \quad \mathrm{h} : \mathrm{Bool}, \quad \mathrm{k} : \mathrm{String} \; \right\} \; \right\}$$

Equações:

$$\{\{\ (\mathrm{f}, \mathrm{Int})\,,\quad (\mathrm{h}, \mathrm{Bool})\,,\quad (\mathrm{k}, \mathrm{String})\ \} \equiv (\{\ (\mathrm{h}, \mathrm{Int})\,,\quad (\mathrm{k}, \mathrm{String})\ \} \ominus \emptyset) \oplus \{\ (\mathrm{f}, \mathrm{Int})\ \}\}$$

Redução:

$$\left(\left\{\left\{\begin{array}{c}(\mathrm{f}, \mathrm{Int})\,, (\mathrm{h}, \mathrm{Bool})\,,\\ (\mathrm{k}, \mathrm{String})\end{array}\right\} \equiv (\{\ (\mathrm{h}, \mathrm{Int})\,,\quad (\mathrm{k}, \mathrm{String})\ \} \ominus \emptyset) \oplus \{\ (\mathrm{f}, \mathrm{Int})\ \}\right\},\ \mathrm{id},\ \emptyset\right) \xrightarrow{2} \bot$$

O conjunto não é satisfazível, pois temos que o mgu de

$$\{(\{\ \mathrm{f} : \mathrm{Int},\quad \mathrm{h} : \mathrm{Bool},\quad \mathrm{k} : \mathrm{String}\ \},\ \{\ \mathrm{f} : \mathrm{Int},\quad \mathrm{h} : \mathrm{Int},\quad \mathrm{k} : \mathrm{String}\ \})\}$$

não é definido, pois os tipos Bool e Int não são unificáveis.

Exemplo A.6. Conjunto de restrições:

$$\{\ +\mathrm{f} :: [\alpha_1] \to \alpha_2 \to \{\ \mathrm{f} : [\mathrm{Int}]\,,\quad \mathrm{h} : \mathrm{Int},\quad \mathrm{k} : \mathrm{String}\ \}\ \}$$

Equações:

$$\{\{\ (\mathrm{f}, [\mathrm{Int}])\,,\quad (\mathrm{h}, \mathrm{Int})\,,\quad (\mathrm{k}, \mathrm{String})\ \} \equiv (r_{\alpha_2} \ominus \emptyset) \oplus \{\ (\mathrm{f}, [\alpha_1])\ \}\}$$

Redução:

$$(\{\{\ (\mathrm{f}, [\mathrm{Int}])\,,\quad (\mathrm{h}, \mathrm{Int})\,,\quad (\mathrm{k}, \mathrm{String})\ \} \equiv (r_{\alpha_2} \ominus \emptyset) \oplus \{\ (\mathrm{f}, [\alpha_1])\ \}\},\ \mathrm{id},\ \emptyset) \xrightarrow{3}$$

$$\text{onde}\quad S = [\alpha_1 \mapsto \mathrm{Int}, \alpha_2 \mapsto \{\ \mathrm{h} : \mathrm{Int},\quad \mathrm{k} : \mathrm{String}\ \}] \qquad (\emptyset,\ S,\ \emptyset)$$

O conjunto de restrições é satisfazível, sendo que a substituição principal é igual a:

$$[\alpha_1 \mapsto \mathrm{Int}, \alpha_2 \mapsto \{\ \mathrm{h} : \mathrm{Int},\quad \mathrm{k} : \mathrm{String}\ \}]$$

Exemplo A.7. Conjunto de restrições:

$$\{\ +\mathrm{f} :: \mathrm{Int} \to \alpha_1 \to \{\ \mathrm{h} : \mathrm{Int},\quad \mathrm{k} : \mathrm{String}\ \}\ \}$$

Equações:

$$\{\{\ (\mathrm{h}, \mathrm{Int})\,,\quad (\mathrm{k}, \mathrm{String})\ \} \equiv (r_{\alpha_1} \ominus \emptyset) \oplus \{\ (\mathrm{f}, \mathrm{Int})\ \}\}$$

Redução:

$$(\{\{\ (\mathrm{h}, \mathrm{Int})\,,\quad (\mathrm{k}, \mathrm{String})\ \} \equiv (r_{\alpha_1} \ominus \emptyset) \oplus \{\ (\mathrm{f}, \mathrm{Int})\ \}\},\ \mathrm{id},\ \emptyset) \xrightarrow{3} \bot$$

O conjunto não é satisfazível, pois: $\{\mathrm{f}\} \not\subseteq \{\mathrm{h}, \mathrm{k}\}$.

Exemplo A.8. Conjunto de restrições:

$$\left\{ \text{ff} :: \alpha_1 \to \alpha_2 \to \alpha_3, \quad +\text{f} :: \alpha_1 \to \alpha_3 \to \alpha_2 \right\}$$

Equações:

$$\left\{ r_{\alpha_2} \equiv (r_{\alpha_3} \ominus \emptyset) \oplus \left\{ \ (\text{f}, \alpha_1) \ \right\} \right\}$$

Redução:

$$\left(\left\{ r_{\alpha_2} \equiv (r_{\alpha_3} \ominus \emptyset) \oplus \left\{ \ (\text{f}, \alpha_1) \ \right\} \right\}, \text{id}, \emptyset \right) \xrightarrow{4}$$

$$\left(\left\{ r_{\alpha_3} \equiv \left(r_{\alpha_2} \ominus \left\{ \ (\text{f}, \alpha_1) \ \right\} \right) \oplus \emptyset \right\}, \text{id}, \emptyset \right)$$

O conjunto é satisfazível, com a substituição identidade como substituição principal.

Exemplo A.9. Conjunto de restrições:

$$\left\{ \begin{array}{ll} \text{ff} :: \alpha_1 \to \alpha_2 \to \alpha_3 \to \alpha_4 \to \alpha_5, & +\text{f} :: \alpha_6 \to \alpha_1 \to \alpha_4, \\ +\text{g} :: \alpha_7 \to \alpha_4 \to \alpha_3, & +\text{g} :: \alpha_8 \to \alpha_2 \to \alpha_5, \\ +\text{f} :: \alpha_9 \to \alpha_5 \to \alpha_3 & \end{array} \right\}$$

Equações:

$$\left\{ \begin{array}{lcl} r_{\alpha_4} & \equiv & (r_{\alpha_1} \ominus \emptyset) \oplus \left\{ \ (\text{f}, \alpha_6) \ \right\} \\ r_{\alpha_3} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \left\{ \ (\text{g}, \alpha_7) \ \right\} \\ r_{\alpha_5} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \left\{ \ (\text{g}, \alpha_8) \ \right\} \\ r_{\alpha_3} & \equiv & (r_{\alpha_5} \ominus \emptyset) \oplus \left\{ \ (\text{f}, \alpha_9) \ \right\} \end{array} \right\}$$

Redução:

$$\left(\left\{\begin{array}{lcl} r_{\alpha_4} & \equiv & (r_{\alpha_1} \ominus \emptyset) \oplus \{\ (\mathrm{f}, \alpha_6)\ \} \\ \underline{r_{\alpha_3}} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \{\ (\mathrm{g}, \alpha_7)\ \} \\ r_{\alpha_5} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \{\ (\mathrm{g}, \alpha_8)\ \} \\ r_{\alpha_3} & \equiv & (r_{\alpha_5} \ominus \emptyset) \oplus \{\ (\mathrm{f}, \alpha_9)\ \} \end{array}\right\}, \mathrm{id}, \emptyset\right) \xrightarrow{4}$$

$$\left(\left\{\begin{array}{lcl} r_{\alpha_4} & \equiv & (r_{\alpha_1} \ominus \emptyset) \oplus \{\ (\mathrm{f}, \alpha_6)\ \} \\ r_{\alpha_5} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \{\ (\mathrm{g}, \alpha_8)\ \} \\ \underline{r_{\alpha_3}} & \equiv & (r_{\alpha_5} \ominus \emptyset) \oplus \{\ (\mathrm{f}, \alpha_9)\ \} \\ \rhd\ r_{\alpha_4} & \equiv & \big(r_{\alpha_3} \ominus \{\ (\mathrm{g}, \alpha_7)\ \}\big) \oplus \emptyset \end{array}\right\}, \mathrm{id}, \emptyset\right) \xrightarrow{4}$$

$$\left(\left\{\begin{array}{lcl} \underline{r_{\alpha_4}} & \equiv & (r_{\alpha_1} \ominus \emptyset) \oplus \{\ (\mathrm{f}, \alpha_6)\ \} \\ r_{\alpha_5} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \{\ (\mathrm{g}, \alpha_8)\ \} \\ \underline{r_{\alpha_4}} & \equiv & \big(r_{\alpha_3} \ominus \{\ (\mathrm{g}, \alpha_7)\ \}\big) \oplus \emptyset \\ \rhd\ r_{\alpha_5} & \equiv & \big(r_{\alpha_3} \ominus \{\ (\mathrm{f}, \alpha_9)\ \}\big) \oplus \emptyset \end{array}\right\}, \mathrm{id}, \emptyset\right) \xrightarrow{8}$$

$$\left(\left\{\begin{array}{lcl} \underline{r_{\alpha_5}} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \{\ (\mathrm{g}, \alpha_8)\ \} \\ \underline{r_{\alpha_5}} & \equiv & \big(r_{\alpha_3} \ominus \{\ (\mathrm{f}, \alpha_9)\ \}\big) \oplus \emptyset \\ \rhd\ r_{\alpha_4} & \equiv & \big(r_{\alpha_3} \ominus \{\ (\mathrm{g}, \alpha_7)\ \}\big) \oplus \emptyset \\ \rhd\ r_{\alpha_3} & \equiv & (r_{\alpha_1} \ominus \emptyset) \oplus \{\ (\mathrm{f}, \alpha_6)\,,\ \ (\mathrm{g}, \alpha_7)\ \} \end{array}\right\}, \mathrm{id}, \emptyset\right) \xrightarrow{8}$$

$$\left(\left\{\begin{array}{lcl} \underline{r_{\alpha_4}} & \equiv & \big(r_{\alpha_3} \ominus \{\ (\mathrm{g}, \alpha_7)\ \}\big) \oplus \emptyset \\ \underline{r_{\alpha_3}} & \equiv & (r_{\alpha_1} \ominus \emptyset) \oplus \{\ (\mathrm{f}, \alpha_6)\,,\ \ (\mathrm{g}, \alpha_7)\ \} \\ \rhd\ r_{\alpha_5} & \equiv & \big(r_{\alpha_3} \ominus \{\ (\mathrm{f}, \alpha_9)\ \}\big) \oplus \emptyset \\ \rhd\ r_{\alpha_3} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \{\ (\mathrm{f}, \alpha_9)\,,\ \ (\mathrm{g}, \alpha_8)\ \} \end{array}\right\}, \mathrm{id}, \emptyset\right) \xrightarrow{11}$$

$$\left(\left\{\begin{array}{rcl} \underline{r_{\alpha_5}} & \equiv & \underline{\left(r_{\alpha_3} \ominus \left\{\ (\mathrm{f}, \alpha_9)\ \right\}\right) \oplus \emptyset} \\ \underline{r_{\alpha_3}} & \equiv & \underline{(r_{\alpha_2} \ominus \emptyset) \oplus \left\{\ (\mathrm{f}, \alpha_9)\,,\ \ (\mathrm{g}, \alpha_8)\ \right\}} \\ \rhd\ r_{\alpha_4} & \equiv & (r_{\alpha_1} \ominus \emptyset) \oplus \left\{\ (\mathrm{f}, \alpha_6)\ \right\} \\ \rhd\ r_{\alpha_3} & \equiv & (r_{\alpha_1} \ominus \emptyset) \oplus \left\{\ (\mathrm{f}, \alpha_6)\,,\ \ (\mathrm{g}, \alpha_7)\ \right\} \end{array}\right\},\ \mathrm{id},\ \emptyset\right) \xrightarrow{11}$$

$$\left(\left\{\begin{array}{rcl} r_{\alpha_4} & \equiv & (r_{\alpha_1} \ominus \emptyset) \oplus \left\{\ (\mathrm{f}, \alpha_6)\ \right\} \\ \underline{r_{\alpha_3}} & \equiv & \underline{(r_{\alpha_1} \ominus \emptyset) \oplus \left\{\ (\mathrm{f}, \alpha_6)\,,\ \ (\mathrm{g}, \alpha_7)\ \right\}} \\ \rhd\ r_{\alpha_5} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \left\{\ (\mathrm{g}, \alpha_8)\ \right\} \\ \rhd\ \underline{r_{\alpha_3}} & \equiv & \underline{(r_{\alpha_2} \ominus \emptyset) \oplus \left\{\ (\mathrm{f}, \alpha_9)\,,\ \ (\mathrm{g}, \alpha_8)\ \right\}} \end{array}\right\},\ \mathrm{id},\ \emptyset\right) \xrightarrow{8}$$

$$\left(\left\{\begin{array}{rcl} r_{\alpha_4} & \equiv & (r_{\alpha_1} \ominus \emptyset) \oplus \left\{\ (\mathrm{f}, \alpha_9)\ \right\} \\ r_{\alpha_5} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \left\{\ (\mathrm{g}, \alpha_8)\ \right\} \\ \rhd\ r_{\alpha_3} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \left\{\ (\mathrm{f}, \alpha_9)\,,\ \ (\mathrm{g}, \alpha_8)\ \right\} \\ \rhd\ \underline{r_{\alpha_2}} & \equiv & \underline{(r_{\alpha_1} \ominus \emptyset) \oplus \emptyset} \end{array}\right\},\ S,\ \emptyset\right) \xrightarrow{5}$$

onde $S = [\alpha_6 \mapsto \alpha_9, \alpha_7 \mapsto \alpha_8]$

$$\left(\left\{\begin{array}{rcl} r_{\alpha_4} & \equiv & (r_{\alpha_1} \ominus \emptyset) \oplus \left\{\ (\mathrm{f}, \alpha_9)\ \right\} \\ r_{\alpha_5} & \equiv & (r_{\alpha_1} \ominus \emptyset) \oplus \left\{\ (\mathrm{g}, \alpha_8)\ \right\} \\ r_{\alpha_3} & \equiv & (r_{\alpha_1} \ominus \emptyset) \oplus \left\{\ (\mathrm{f}, \alpha_9)\,,\ \ (\mathrm{g}, \alpha_8)\ \right\} \end{array}\right\},\ S,\ \{\alpha_1\}\right)$$

onde $S = [\alpha_2 \mapsto \alpha_1, \alpha_6 \mapsto \alpha_9, \alpha_7 \mapsto \alpha_8]$

O conjunto é satisfazível e possui como substituição principal:

$$[\alpha_2 \mapsto \alpha_1, \alpha_6 \mapsto \alpha_9, \alpha_7 \mapsto \alpha_8]$$

Exemplo A.10. Conjunto de restrições:

$$\left\{\begin{array}{ll} \mathrm{ff} :: \alpha_1 \to \alpha_2 \to \alpha_3 \to \alpha_4 \to \alpha_5, & +\mathrm{f} :: \mathrm{Int} \to \alpha_2 \to \alpha_1, \\ +\mathrm{f} :: \alpha_6 \to \alpha_2 \to \alpha_3, & +\mathrm{g} :: \alpha_3 \to \alpha_4 \to \alpha_5, \\ +\mathrm{g} :: \alpha_1 \to \alpha_7 \to \alpha_5 & \end{array}\right\}$$

Equações:

$$\left\{\begin{array}{rcl} r_{\alpha_1} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \left\{\ (\mathrm{f}, \mathrm{Int})\ \right\} \\ r_{\alpha_3} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \left\{\ (\mathrm{f}, \alpha_6)\ \right\} \\ r_{\alpha_5} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \left\{\ (\mathrm{g}, \alpha_3)\ \right\} \\ r_{\alpha_5} & \equiv & (r_{\alpha_7} \ominus \emptyset) \oplus \left\{\ (\mathrm{g}, \alpha_1)\ \right\} \end{array}\right\}$$

Redução:

$$\left(\left\{\begin{array}{lcl} \underline{r_{\alpha_1}} & \equiv & \overline{(r_{\alpha_2} \ominus \emptyset) \oplus \{\ (\mathrm{f}, \mathrm{Int})\ \}} \\ r_{\alpha_3} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \{\ (\mathrm{f}, \alpha_6)\ \} \\ r_{\alpha_5} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \{\ (\mathrm{g}, \alpha_3)\ \} \\ r_{\alpha_5} & \equiv & (r_{\alpha_7} \ominus \emptyset) \oplus \{\ (\mathrm{g}, \alpha_1)\ \} \end{array}\right\}, \mathrm{id}, \emptyset\right) \xrightarrow{4}$$

$$\left(\left\{\begin{array}{lcl} r_{\alpha_3} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \{\ (\mathrm{f}, \alpha_6)\ \} \\ r_{\alpha_5} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \{\ (\mathrm{g}, \alpha_3)\ \} \\ \underline{r_{\alpha_5}} & \equiv & \overline{(r_{\alpha_7} \ominus \emptyset) \oplus \{\ (\mathrm{g}, \alpha_1)\ \}} \\ \rhd\ r_{\alpha_2} & \equiv & \left(r_{\alpha_1} \ominus \{\ (\mathrm{f}, \mathrm{Int})\ \}\right) \oplus \emptyset \end{array}\right\}, \mathrm{id}, \emptyset\right) \xrightarrow{4}$$

$$\left(\left\{\begin{array}{lcl} \underline{r_{\alpha_3}} & \equiv & \overline{(r_{\alpha_2} \ominus \emptyset) \oplus \{\ (\mathrm{f}, \alpha_6)\ \}} \\ r_{\alpha_5} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \{\ (\mathrm{g}, \alpha_3)\ \} \\ \underline{r_{\alpha_2}} & \equiv & \overline{\left(r_{\alpha_1} \ominus \{\ (\mathrm{f}, \mathrm{Int})\ \}\right) \oplus \emptyset} \\ \rhd\ r_{\alpha_7} & \equiv & \left(r_{\alpha_5} \ominus \{\ (\mathrm{g}, \alpha_1)\ \}\right) \oplus \emptyset \end{array}\right\}, \mathrm{id}, \emptyset\right) \xrightarrow{11}$$

$$\left(\left\{\begin{array}{lcl} \underline{r_{\alpha_5}} & \equiv & \overline{(r_{\alpha_4} \ominus \emptyset) \oplus \{\ (\mathrm{g}, \alpha_3)\ \}} \\ \underline{r_{\alpha_7}} & \equiv & \overline{\left(r_{\alpha_5} \ominus \{\ (\mathrm{g}, \alpha_1)\ \}\right) \oplus \emptyset} \\ \rhd\ r_{\alpha_3} & \equiv & \left(r_{\alpha_1} \ominus \{\ (\mathrm{f}, \mathrm{Int})\ \}\right) \oplus \{\ (\mathrm{f}, \alpha_6)\ \} \\ \rhd\ r_{\alpha_2} & \equiv & \left(r_{\alpha_1} \ominus \{\ (\mathrm{f}, \mathrm{Int})\ \}\right) \oplus \emptyset \end{array}\right\}, \mathrm{id}, \emptyset\right) \xrightarrow{11}$$

$$\left(\left\{\begin{array}{lcl} \underline{r_{\alpha_1}} & \equiv & \overline{\left(r_{\alpha_1} \ominus \{\ (\mathrm{f}, \mathrm{Int})\ \}\right) \oplus \{\ (\mathrm{f}, \alpha_6)\ \}} \\ r_{\alpha_2} & \equiv & \left(r_{\alpha_1} \ominus \{\ (\mathrm{f}, \mathrm{Int})\ \}\right) \oplus \emptyset \\ \rhd\ r_{\alpha_7} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \emptyset \\ r_{\alpha_5} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \{\ (\mathrm{g}, \alpha_1)\ \} \end{array}\right\}, [\alpha_3 \mapsto \alpha_1], \emptyset\right) \xrightarrow{6}$$

$$\left(\left\{\begin{array}{lcl} r_{\alpha_2} & \equiv & \left(r_{\alpha_1} \ominus \{\ (\mathrm{f}, \mathrm{Int})\ \}\right) \oplus \emptyset \\ \underline{r_{\alpha_7}} & \equiv & \overline{(r_{\alpha_4} \ominus \emptyset) \oplus \emptyset} \\ r_{\alpha_5} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \{\ (\mathrm{g}, \alpha_1)\ \} \end{array}\right\}, S, \emptyset\right) \xrightarrow{5}$$

$$\text{onde} \quad S = [\alpha_3 \mapsto \alpha_1, \alpha_6 \mapsto \mathrm{Int}]$$

$$\left(\left\{\begin{array}{lcl} r_{\alpha_2} & \equiv & \left(r_{\alpha_1} \ominus \{\ (\mathrm{f}, \mathrm{Int})\ \}\right) \oplus \emptyset \\ r_{\alpha_5} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \{\ (\mathrm{g}, \alpha_1)\ \} \end{array}\right\}, S, \{\alpha_4\}\right)$$

$$\text{onde} \quad S = [\alpha_3 \mapsto \alpha_1, \alpha_6 \mapsto \mathrm{Int}, \alpha_7 \mapsto \alpha_4]$$

O conjunto é satisfazível, com substituição principal igual a:

$$[\alpha_3 \mapsto \alpha_1, \alpha_6 \mapsto \mathrm{Int}, \alpha_7 \mapsto \alpha_4]$$

Exemplo A.11. Conjunto de restrições:

$$\left\{ \begin{array}{ll} \mathrm{ff} :: \alpha_1 \to \alpha_2 \to \alpha_3 \to \alpha_4 \to \alpha_5, & +\mathrm{g} :: \mathrm{Int} \to \alpha_2 \to \alpha_1, \\ +\mathrm{f} :: \alpha_6 \to \alpha_2 \to \alpha_3, & +\mathrm{g} :: \alpha_3 \to \alpha_4 \to \alpha_5, \\ +\mathrm{g} :: \alpha_1 \to \alpha_7 \to \alpha_5 & \end{array} \right\}$$

Equações:

$$\left\{ \begin{array}{lcl} r_{\alpha_1} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \left\{ \ (\mathrm{g}, \mathrm{Int}) \ \right\} \\ r_{\alpha_3} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \alpha_6) \ \right\} \\ r_{\alpha_5} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \left\{ \ (\mathrm{g}, \alpha_3) \ \right\} \\ r_{\alpha_5} & \equiv & (r_{\alpha_7} \ominus \emptyset) \oplus \left\{ \ (\mathrm{g}, \alpha_1) \ \right\} \end{array} \right\}$$

Redução:

$$\left(\left\{ \begin{array}{lcl} \underline{r_{\alpha_1}} & \equiv & \underline{(r_{\alpha_2} \ominus \emptyset) \oplus \left\{ \ (\mathrm{g}, \mathrm{Int}) \ \right\}} \\ r_{\alpha_3} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \alpha_6) \ \right\} \\ r_{\alpha_5} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \left\{ \ (\mathrm{g}, \alpha_3) \ \right\} \\ r_{\alpha_5} & \equiv & (r_{\alpha_7} \ominus \emptyset) \oplus \left\{ \ (\mathrm{g}, \alpha_1) \ \right\} \end{array} \right\}, \mathrm{id}, \emptyset \right) \xrightarrow{4}$$

$$\left(\left\{ \begin{array}{lcl} r_{\alpha_3} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \alpha_6) \ \right\} \\ r_{\alpha_5} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \left\{ \ (\mathrm{g}, \alpha_3) \ \right\} \\ \underline{r_{\alpha_5}} & \equiv & \underline{(r_{\alpha_7} \ominus \emptyset) \oplus \left\{ \ (\mathrm{g}, \alpha_1) \ \right\}} \\ \rhd\ r_{\alpha_2} & \equiv & \left(r_{\alpha_1} \ominus \left\{ \ (\mathrm{g}, \mathrm{Int}) \ \right\}\right) \oplus \emptyset \end{array} \right\}, \mathrm{id}, \emptyset \right) \xrightarrow{4}$$

$$\left(\left\{ \begin{array}{lcl} \underline{r_{\alpha_3}} & \equiv & \underline{(r_{\alpha_2} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \alpha_6) \ \right\}} \\ r_{\alpha_5} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \left\{ \ (\mathrm{g}, \alpha_3) \ \right\} \\ \underline{r_{\alpha_2}} & \equiv & \underline{\left(r_{\alpha_1} \ominus \left\{ \ (\mathrm{g}, \mathrm{Int}) \ \right\}\right) \oplus \emptyset} \\ \rhd\ r_{\alpha_7} & \equiv & \left(r_{\alpha_5} \ominus \left\{ \ (\mathrm{g}, \alpha_1) \ \right\}\right) \oplus \emptyset \end{array} \right\}, \mathrm{id}, \emptyset \right) \xrightarrow{11}$$

$$\left(\left\{ \begin{array}{lcl} \underline{r_{\alpha_5}} & \equiv & \underline{(r_{\alpha_4} \ominus \emptyset) \oplus \left\{ \ (\mathrm{g}, \alpha_3) \ \right\}} \\ \underline{r_{\alpha_7}} & \equiv & \underline{\left(r_{\alpha_5} \ominus \left\{ \ (\mathrm{g}, \alpha_1) \ \right\}\right) \oplus \emptyset} \\ \rhd\ r_{\alpha_3} & \equiv & \left(r_{\alpha_1} \ominus \left\{ \ (\mathrm{g}, \mathrm{Int}) \ \right\}\right) \oplus \left\{ \ (\mathrm{f}, \alpha_6) \ \right\} \\ \rhd\ r_{\alpha_2} & \equiv & \left(r_{\alpha_1} \ominus \left\{ \ (\mathrm{g}, \mathrm{Int}) \ \right\}\right) \oplus \emptyset \end{array} \right\}, \mathrm{id}, \emptyset \right) \xrightarrow{11}$$

$$\left(\left\{ \begin{array}{lcl} \underline{r_{\alpha_1}} & \equiv & \underline{\left(r_{\alpha_1} \ominus \left\{ \ (\mathrm{g}, \mathrm{Int}) \ \right\}\right) \oplus \left\{ \ (\mathrm{f}, \alpha_6) \ \right\}} \\ r_{\alpha_2} & \equiv & \left(r_{\alpha_1} \ominus \left\{ \ (\mathrm{g}, \mathrm{Int}) \ \right\}\right) \oplus \emptyset \\ \rhd\ r_{\alpha_7} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \emptyset \\ r_{\alpha_5} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \left\{ \ (\mathrm{g}, \alpha_1) \ \right\} \end{array} \right\}, [\alpha_3 \mapsto \alpha_1], \emptyset \right) \xrightarrow{6} \bot$$

O conjunto não é satisfazível, pois $\{\mathrm{g}\} \neq \{\mathrm{f}\}$.

Exemplo A.12. Conjunto de restrições:

$$\left\{ \begin{array}{ll} \mathrm{ff} :: \alpha_1 \to \alpha_2 \to \alpha_3 \to \alpha_4 \to \alpha_5, & +\mathrm{f} :: \mathrm{Int} \to \alpha_2 \to \alpha_1, \\ +\mathrm{f} :: \mathrm{String} \to \alpha_2 \to \alpha_3, & +\mathrm{g} :: \alpha_3 \to \alpha_4 \to \alpha_5, \\ +\mathrm{g} :: \alpha_1 \to \alpha_6 \to \alpha_5 & \end{array} \right\}$$

Equações:

$$\left\{ \begin{array}{lcl} r_{\alpha_1} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \mathrm{Int}) \ \right\} \\ r_{\alpha_3} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \mathrm{String}) \ \right\} \\ r_{\alpha_5} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \left\{ \ (\mathrm{g}, \alpha_3) \ \right\} \\ r_{\alpha_5} & \equiv & (r_{\alpha_6} \ominus \emptyset) \oplus \left\{ \ (\mathrm{g}, \alpha_1) \ \right\} \end{array} \right\}$$

Redução:

$$\left(\left\{ \begin{array}{lcl} \underline{r_{\alpha_1}} & \equiv & \overline{(r_{\alpha_2} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \mathrm{Int}) \ \right\}} \\ r_{\alpha_3} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \mathrm{String}) \ \right\} \\ r_{\alpha_5} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \left\{ \ (\mathrm{g}, \alpha_3) \ \right\} \\ r_{\alpha_5} & \equiv & (r_{\alpha_6} \ominus \emptyset) \oplus \left\{ \ (\mathrm{g}, \alpha_1) \ \right\} \end{array} \right\}, \mathrm{id}, \emptyset \right) \xrightarrow{4}$$

$$\left(\left\{ \begin{array}{lcl} r_{\alpha_3} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \mathrm{String}) \ \right\} \\ r_{\alpha_5} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \left\{ \ (\mathrm{g}, \alpha_3) \ \right\} \\ \underline{r_{\alpha_5}} & \equiv & \overline{(r_{\alpha_6} \ominus \emptyset) \oplus \left\{ \ (\mathrm{g}, \alpha_1) \ \right\}} \\ \rhd\ r_{\alpha_2} & \equiv & \overline{\left(r_{\alpha_1} \ominus \left\{ \ (\mathrm{f}, \mathrm{Int}) \ \right\}\right) \oplus \emptyset} \end{array} \right\}, \mathrm{id}, \emptyset \right) \xrightarrow{4}$$

$$\left(\left\{ \begin{array}{lcl} \underline{r_{\alpha_3}} & \equiv & \overline{(r_{\alpha_2} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \mathrm{String}) \ \right\}} \\ r_{\alpha_5} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \left\{ \ (\mathrm{g}, \alpha_3) \ \right\} \\ \underline{r_{\alpha_2}} & \equiv & \overline{\left(r_{\alpha_1} \ominus \left\{ \ (\mathrm{f}, \mathrm{Int}) \ \right\}\right) \oplus \emptyset} \\ \rhd\ r_{\alpha_6} & \equiv & \overline{\left(r_{\alpha_5} \ominus \left\{ \ (\mathrm{g}, \alpha_1) \ \right\}\right) \oplus \emptyset} \end{array} \right\}, \mathrm{id}, \emptyset \right) \xrightarrow{11}$$

$$\left(\left\{ \begin{array}{lcl} \underline{r_{\alpha_5}} & \equiv & \overline{(r_{\alpha_4} \ominus \emptyset) \oplus \left\{ \ (\mathrm{g}, \alpha_3) \ \right\}} \\ \underline{r_{\alpha_6}} & \equiv & \overline{\left(r_{\alpha_5} \ominus \left\{ \ (\mathrm{g}, \alpha_1) \ \right\}\right) \oplus \emptyset} \\ \rhd\ r_{\alpha_3} & \equiv & \left(r_{\alpha_1} \ominus \left\{ \ (\mathrm{f}, \mathrm{Int}) \ \right\}\right) \oplus \left\{ \ (\mathrm{f}, \mathrm{String}) \ \right\} \\ \rhd\ r_{\alpha_2} & \equiv & \left(r_{\alpha_1} \ominus \left\{ \ (\mathrm{f}, \mathrm{Int}) \ \right\}\right) \oplus \emptyset \end{array} \right\}, \mathrm{id}, \emptyset \right) \xrightarrow{11}$$

$$\left(\left\{ \begin{array}{lcl} \underline{r_{\alpha_1}} & \equiv & \overline{\left(r_{\alpha_1} \ominus \left\{ \ (\mathrm{f}, \mathrm{Int}) \ \right\}\right) \oplus \left\{ \ (\mathrm{f}, \mathrm{String}) \ \right\}} \\ r_{\alpha_2} & \equiv & \left(r_{\alpha_1} \ominus \left\{ \ (\mathrm{f}, \mathrm{Int}) \ \right\}\right) \oplus \emptyset \\ \rhd\ r_{\alpha_6} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \emptyset \\ r_{\alpha_5} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \left\{ \ (\mathrm{g}, \alpha_1) \ \right\} \end{array} \right\}, [\alpha_3 \mapsto \alpha_1], \emptyset \right) \xrightarrow{6} \bot$$

O conjunto não é satisfazível, pois $\mathrm{mgu}\,(\{(\mathrm{Int}, \mathrm{String})\}) = \bot$.

Exemplo A.13. Conjunto de restrições:

$$\left\{ \ +\mathrm{f} :: \mathrm{Int} \to \alpha_1 \to \alpha_2, \quad +\mathrm{f} :: \mathrm{Int} \to \alpha_1 \to \alpha_3 \ \right\}$$

Equações:

$$\left\{ \begin{array}{lcl} r_{\alpha_2} & \equiv & (r_{\alpha_1} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \mathrm{Int}) \ \right\} \\ r_{\alpha_3} & \equiv & (r_{\alpha_1} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \mathrm{Int}) \ \right\} \end{array} \right\}$$

Redução:

$$\left(\left\{ \begin{array}{lcl} \underline{r_{\alpha_2}} & \equiv & \underline{(r_{\alpha_1} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \mathrm{Int}) \ \right\}} \\ \underline{r_{\alpha_3}} & \equiv & \underline{(r_{\alpha_1} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \mathrm{Int}) \ \right\}} \end{array} \right\}, \mathrm{id}, \emptyset \right) \xrightarrow{10}$$

$$\left(\left\{ r_{\alpha_2} \equiv (r_{\alpha_1} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \mathrm{Int}) \ \right\} \right\}, [\alpha_3 \mapsto \alpha_2], \emptyset \right)$$

Satisfazível, com substituição principal igual a:

$$[\alpha_3 \mapsto \alpha_2]$$

Exemplo A.14. Conjunto de restrições:

$$\left\{ \begin{array}{l} \mathrm{f} :: \alpha_1 \to \alpha_2 \to \alpha_3 \to \alpha_4, \quad +\mathrm{f} :: \alpha_5 \to \alpha_1 \to \alpha_3, \\ +\mathrm{f} :: \mathrm{Int} \to \alpha_2 \to \alpha_3 \end{array} \right\}$$

Equações:

$$\left\{ \begin{array}{lcl} r_{\alpha_3} & \equiv & (r_{\alpha_1} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \alpha_5) \ \right\} \\ r_{\alpha_3} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \mathrm{Int}) \ \right\} \end{array} \right\}$$

Redução:

$$\left(\left\{ \begin{array}{lcl} \underline{r_{\alpha_3}} & \equiv & \underline{(r_{\alpha_1} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \alpha_5) \ \right\}} \\ \underline{r_{\alpha_3}} & \equiv & \underline{(r_{\alpha_2} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \mathrm{Int}) \ \right\}} \end{array} \right\}, \mathrm{id}, \emptyset \right) \xrightarrow{8}$$

$$\left(\left\{ \begin{array}{lcl} \rhd r_{\alpha_3} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \mathrm{Int}) \ \right\} \\ \rhd \underline{r_{\alpha_2}} & \equiv & \underline{(r_{\alpha_1} \ominus \emptyset) \oplus \emptyset} \end{array} \right\}, [\alpha_5 \mapsto \mathrm{Int}], \emptyset \right) \xrightarrow{5}$$

$$\begin{array}{c} \left(\left\{ r_{\alpha_3} \equiv (r_{\alpha_1} \ominus \emptyset) \oplus \left\{ \ (\mathrm{f}, \mathrm{Int}) \ \right\} \right\}, S, \{\alpha_1\} \right) \\ \text{onde} \quad S = [\alpha_2 \mapsto \alpha_1, \alpha_5 \mapsto \mathrm{Int}] \end{array}$$

O conjunto é satisfazível, com substituição principal igual a:

$$[\alpha_2 \mapsto \alpha_1, \alpha_5 \mapsto \mathrm{Int}]$$

Exemplo A.15. Tipo restringido:

$$\left\{ \begin{array}{ll} +\mathrm{f} :: \alpha_1 \to \alpha_2 \to \alpha_3, & +\mathrm{f} :: \mathrm{String} \to \alpha_2 \to \alpha_4, \\ +\mathrm{g} :: \mathrm{Bool} \to \alpha_4 \to \alpha_5, & +\mathrm{h} :: \alpha_6 \to \alpha_7 \to \alpha_5 \end{array} \right\} . \alpha_3 \to \alpha_3 \to \alpha_7$$

Equações:

$$\left\{ \begin{array}{lcl} r_{\alpha_3} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \{\ (\mathrm{f}, \alpha_1)\ \} \\ r_{\alpha_4} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \{\ (\mathrm{f}, \mathrm{String})\ \} \\ r_{\alpha_5} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \{\ (\mathrm{g}, \mathrm{Bool})\ \} \\ r_{\alpha_5} & \equiv & (r_{\alpha_7} \ominus \emptyset) \oplus \{\ (\mathrm{h}, \alpha_6)\ \} \end{array} \right\}$$

Redução:

$$\left(\left\{ \begin{array}{lcl} \underline{r_{\alpha_3}} & \equiv & \underline{(r_{\alpha_2} \ominus \emptyset) \oplus \{\ (\mathrm{f}, \alpha_1)\ \}} \\ r_{\alpha_4} & \equiv & (r_{\alpha_2} \ominus \emptyset) \oplus \{\ (\mathrm{f}, \mathrm{String})\ \} \\ r_{\alpha_5} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \{\ (\mathrm{g}, \mathrm{Bool})\ \} \\ r_{\alpha_5} & \equiv & (r_{\alpha_7} \ominus \emptyset) \oplus \{\ (\mathrm{h}, \alpha_6)\ \} \end{array} \right\}, \mathrm{id}, \emptyset \right) \xrightarrow{4}$$

$$\left(\left\{ \begin{array}{lcl} \underline{r_{\alpha_4}} & \equiv & \underline{(r_{\alpha_2} \ominus \emptyset) \oplus \{\ (\mathrm{f}, \mathrm{String})\ \}} \\ r_{\alpha_5} & \equiv & (r_{\alpha_4} \ominus \emptyset) \oplus \{\ (\mathrm{g}, \mathrm{Bool})\ \} \\ r_{\alpha_5} & \equiv & (r_{\alpha_7} \ominus \emptyset) \oplus \{\ (\mathrm{h}, \alpha_6)\ \} \\ \rhd\ \underline{r_{\alpha_2}} & \equiv & \underline{\left(r_{\alpha_3} \ominus \{\ (\mathrm{f}, \alpha_1)\ \}\right) \oplus \emptyset} \end{array} \right\}, \mathrm{id}, \emptyset \right) \xrightarrow{11}$$

$$\left(\left\{ \begin{array}{lcl} \underline{r_{\alpha_5}} & \equiv & \underline{(r_{\alpha_4} \ominus \emptyset) \oplus \{\ (\mathrm{g}, \mathrm{Bool})\ \}} \\ \underline{r_{\alpha_5}} & \equiv & \underline{(r_{\alpha_7} \ominus \emptyset) \oplus \{\ (\mathrm{h}, \alpha_6)\ \}} \\ \rhd\ r_{\alpha_4} & \equiv & \left(r_{\alpha_3} \ominus \{\ (\mathrm{f}, \alpha_1)\ \}\right) \oplus \{\ (\mathrm{f}, \mathrm{String})\ \} \\ \rhd\ r_{\alpha_2} & \equiv & \left(r_{\alpha_3} \ominus \{\ (\mathrm{f}, \alpha_1)\ \}\right) \oplus \emptyset \end{array} \right\}, \mathrm{id}, \emptyset \right) \xrightarrow{8}$$

$$\left(\left\{ \begin{array}{lcl} \underline{r_{\alpha_4}} & \equiv & \underline{\left(r_{\alpha_3} \ominus \{\ (\mathrm{f}, \alpha_1)\ \}\right) \oplus \{\ (\mathrm{f}, \mathrm{String})\ \}} \\ r_{\alpha_2} & \equiv & \left(r_{\alpha_3} \ominus \{\ (\mathrm{f}, \alpha_1)\ \}\right) \oplus \emptyset \\ \rhd\ \underline{r_{\alpha_5}} & \equiv & \underline{(r_{\alpha_4} \ominus \emptyset) \oplus \{\ (\mathrm{g}, \mathrm{Bool})\ \}} \\ \rhd\ r_{\alpha_4} & \equiv & \left(r_{\alpha_7} \ominus \{\ (\mathrm{g}, \mathrm{Bool})\ \}\right) \oplus \{\ (\mathrm{h}, \alpha_6)\ \} \end{array} \right\}, \mathrm{id}, \emptyset \right) \xrightarrow{11}$$

$$\left(\left\{\begin{array}{lcl} r_{\alpha_2} & \equiv & \left(r_{\alpha_3} \ominus \left\{\ (\mathrm{f}, \alpha_1)\ \right\}\right) \oplus \emptyset \\ \underline{r_{\alpha_4}} & \equiv & \left(r_{\alpha_7} \ominus \left\{\ (\mathrm{g}, \mathrm{Bool})\ \right\}\right) \oplus \left\{\ (\mathrm{h}, \alpha_6)\ \right\} \\ \rhd\ r_{\alpha_5} & \equiv & \left(r_{\alpha_3} \ominus \left\{\ (\mathrm{f}, \alpha_1)\ \right\}\right) \oplus \left\{\ (\mathrm{f}, \mathrm{String}),\ \ (\mathrm{g}, \mathrm{Bool})\ \right\} \\ \rhd\ \underline{r_{\alpha_4}} & \equiv & \left(r_{\alpha_3} \ominus \left\{\ (\mathrm{f}, \alpha_1)\ \right\}\right) \oplus \left\{\ (\mathrm{f}, \mathrm{String})\ \right\} \end{array}\right\},\ \mathrm{id},\ \emptyset\right) \xrightarrow{8}$$

$$\left(\left\{\begin{array}{lcl} r_{\alpha_2} & \equiv & \left(r_{\alpha_3} \ominus \left\{\ (\mathrm{f}, \alpha_1)\ \right\}\right) \oplus \emptyset \\ r_{\alpha_5} & \equiv & \left(r_{\alpha_3} \ominus \left\{\ (\mathrm{f}, \alpha_1)\ \right\}\right) \oplus \left\{\ (\mathrm{f}, \mathrm{String}),\ \ (\mathrm{g}, \mathrm{Bool})\ \right\} \\ \rhd\ \underline{r_{\alpha_4}} & \equiv & \left(r_{\alpha_7} \ominus \left\{\ (\mathrm{g}, \mathrm{Bool})\ \right\}\right) \oplus \left\{\ (\mathrm{h}, \alpha_6)\ \right\} \\ \rhd\ \underline{r_{\alpha_7}} & \equiv & \left(r_{\alpha_3} \ominus \left\{\ (\mathrm{f}, \alpha_1),\ \ (\mathrm{h}, \alpha_6)\ \right\}\right) \oplus \left\{\ (\mathrm{f}, \mathrm{String}),\ \ (\mathrm{g}, \mathrm{Bool})\ \right\} \end{array}\right\},\ \mathrm{id},\ \emptyset\right) \xrightarrow{11}$$

$$\left(\left\{\begin{array}{lcl} r_{\alpha_2} & \equiv & \left(r_{\alpha_3} \ominus \left\{\ (\mathrm{f}, \alpha_1)\ \right\}\right) \oplus \emptyset \\ r_{\alpha_5} & \equiv & \left(r_{\alpha_3} \ominus \left\{\ (\mathrm{f}, \alpha_1)\ \right\}\right) \oplus \left\{\ (\mathrm{f}, \mathrm{String}),\ \ (\mathrm{g}, \mathrm{Bool})\ \right\} \\ \rhd\ r_{\alpha_4} & \equiv & \left(r_{\alpha_3} \ominus \left\{\ (\mathrm{f}, \alpha_1),\ \ (\mathrm{h}, \alpha_6)\ \right\}\right) \oplus \left\{\ (\mathrm{f}, \mathrm{String}),\ \ (\mathrm{h}, \alpha_6)\ \right\} \\ \rhd\ r_{\alpha_7} & \equiv & \left(r_{\alpha_3} \ominus \left\{\ (\mathrm{f}, \alpha_1),\ \ (\mathrm{h}, \alpha_6)\ \right\}\right) \oplus \left\{\ (\mathrm{f}, \mathrm{String}),\ \ (\mathrm{g}, \mathrm{Bool})\ \right\} \end{array}\right\},\ \mathrm{id},\ \emptyset\right)$$

O conjunto de restrições é satisfazível, com a substituição identidade como substituição principal.

O tipo equivalente usando a sintaxe amigável:

$$\alpha_3 \prec \{\mathrm{f} : \alpha_1,\ -\mathrm{g},\ \mathrm{h} : \alpha_6,\ \ldots\} \to \alpha_3 \to \alpha_3 + \{\mathrm{f} : \mathrm{String},\ +\mathrm{g} : \mathrm{Bool},\ -\mathrm{h}\}$$