Eva Catarina Gomes Maia

Inferência de tipos em Python

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Julho de 2010

Eva Catarina Gomes Maia

Inferência de tipos em Python

Dissertação submetida à Faculdade de Ciências da Universidade do Porto como parte dos requisitos para a obtenção do grau de Mestre em Ciência de Computadores

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Julho de 2010

Aos meus pais.

Ao meu irmão.

Ao Hélder.

Agradecimentos

Em primeiro lugar quero agradecer aos meus orientadores, Nelma Moreira e Rogério Reis, por acreditarem e confiarem no meu trabalho. Agradeço pelo apoio incansável, pela capacidade de ensinar, pela constante disponibilidade, incentivo e paciência. Por tudo, manifesto o meu profundo reconhecimento.

Aos meus colegas, em geral, por todos os momentos partilhados. Em especial, agradeço ao Pedro, ao André e à Catarina pelo apoio, ânimo e pelas sugestões e interesse que sempre demonstraram no meu trabalho.

Agradeço à minha família, em especial aos meus pais e ao meu irmão, que sempre me apoiaram e incentivaram em todas as minhas decisões. Aos meus pais agradeço todo o esforço e sacrifício para me proporcionarem uma boa educação. Ao meu irmão agradeço a amizade, exemplo e ajuda em todos os momentos da minha vida.

Por fim, mas não menos importante, agradeço ao Hélder todo o apoio, compreensão e carinho em todos os momentos, bons e maus, da nossa vida.

Abstract

Nowadays, certification of software security is increasingly important for industry. It is necessary to certificate, from critical systems software, like those used in aviation, to embedded systems, like those used in cellphones. The growing need for security, accuracy and speed of software development of these systems leads to its developed in high-level languages.

In the last thirty years, type systems have been developed and successfully used in different programming languages. A type system is a component of typed languages, which defines a set of rules that associate types to program constructors . Using a type system can prevent the occurrence of certain errors during program execution. In languages where types are explicitly declared the type system allows to type checking the program. In languages in which the types are not explicitly declared, like Python, but for which there is an associated type system, types can be obtained by a process of inference.

Python is a very high-level, objected oriented, programming language. It is dynamically typed, which allows programmers greater flexibility, but deprives them of the advantages of static typing, as early error detection. So to be able to check Python programs, in a static way, we have to choose a subset of the Python language. The subset chosen was RPython, which was informally identified in the scope of the PyPy project.

In this essay we describe a type system, we defined, for RPython language. The rules definition of this type system forced us to formalize RPython language. This is, as

far as we know, its first formal definition. We also describe our implementation of a static type inference algorithm for this language.

Resumo

A certificação da segurança de *software* é hoje cada vez mais importante para a indústria. É necessário certificar desde o *software* de sistemas críticos, como os utilizados na aviação, até ao de sistemas embebidos, como os utilizados nos telemóveis. A crescente necessidade de segurança, correcção e rapidez de desenvolvimento do *software* destes sistemas conduz a que ele seja desenvolvido em linguagens de alto nível.

Nos últimos trinta anos, os sistemas de tipos têm sido desenvolvidos e usados com sucesso em diferentes linguagens de programação. Um sistema de tipos, é um componente das linguagens tipificadas, que define um conjunto de regras que associam tipos aos objectos do programa. O uso de um sistema de tipos permite prevenir a ocorrência de determinados erros durante a execução do programa. Nas linguagens em que os tipos são declarados explicitamente o sistema de tipos permite fazer a verificação de tipos desse programa. Nas linguagens em que os tipos não são declarados explicitamente, como o Python, mas para as quais existe um sistema de tipos associado, os tipos poderão ser obtidos por um processo de inferência.

O Python é uma linguagem de programação de muito alto nível e orientada a objectos. É dinamicamente tipificada, o que permite ao programador uma maior flexibilidade, no entanto priva-o das vantagens da tipificação estática, como a detecção precoce de erros. Assim, para ser possível a verificação, na ausência de execução, de programas Python, temos que escolher um subconjunto da linguagem. O subconjunto escolhido foi o RPython, o qual foi identificado informalmente no âmbito do projecto PyPy.

Nesta dissertação descrevemos um sistema de tipos, que desenvolvemos, para a linguagem RPython. A definição das regras deste sistema de tipos obrigou a formalizar a linguagem RPython, a qual não possuía qualquer definição formal. É também descrita a implementação que efectuamos de um algoritmo de inferência de tipos, na ausência de execução, para esta linguagem.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

A verificação formal de programas é, actualmente, muito importante devido ao aumento da necessidade de certificar o *software* como fiável. Em especial, é importante certificar o *software* para os sistemas críticos e embebidos, e garantir a sua segurança integridade e correcção. As aplicações destes sistemas são normalmente implementadas em C ou Java. No entanto, quando o desempenho destas aplicações não é crítico, a necessidade de segurança, correcção e rapidez de desenvolvimento justificam a utilização de linguagens de alto nível, como o Python.

As linguagens de alto nível são, muitas vezes, dinamicamente tipificadas. Estas, são bastante atractivas uma vez que garantem, em tempo de execução, que nenhum programa correcto é rejeitado prematuramente. Apenas são rejeitados os programas em que ocorrem erros em tempo de execução. Mas esta permissividade acarreta problemas. Os erros que podem ser detectados pela tipificação estática, por exemplo a chamada a uma função em que os argumentos não têm o tipo correcto, quando são detectados apenas em tempo de execução, têm um custo elevado na fiabilidade dos programas. Nas linguagens tipificadas estaticamente os erros são detectados precocemente, e a sua origem é identificada com mais precisão.

O Python [Ros95] é uma linguagem de programação de muito alto nível, orientada a objectos. Possui uma sintaxe clara, que facilita a legibilidade do código e o desenvolvi-

mento rápido de programas. É dinamicamente tipificada e permite que as variáveis tomem diversos tipos, em diferentes locais do programa. Desta forma, para ser possível a verificação de programas Python, é necessário o desenvolvimento de um sistema de verificação de tipos, na ausência de execução. Para tal temos que reduzir a linguagem a um subconjunto onde este tipo de verificação seja possível. No âmbito do projecto europeu Pypy [Pro], cujo objectivo é a possibilidade de execução eficiente de Python e a construção de um compilador *Just-in-time*, foi identificado (informalmente) um subconjunto do Python (RPython) para o qual é possível inferir tipos em tempo de compilação.

1.1 Motivação do trabalho

O objectivo do nosso trabalho era criar uma ferramenta que realizasse a inferência de tipos em Python, na ausência de execução. No entanto, como qualquer outra linguagem dinamicamente tipificada, o Python possui algumas características que dificultam a inferência de tipos, na ausência de execução. Deste modo, reduzimos a linguagem alvo da inferência de tipos a um subconjunto do Python. O subconjunto que escolhemos foi o RPython.

Nas linguagens em que, como no Python, não se declara explicitamente o tipo das variáveis ou funções, mas que possuem um sistema de tipos associado, os tipos poderão ser obtidos por um processo de inferência. Assim, para concretizar o nosso objectivo, definimos um sistema de tipos para a linguagem RPython, o qual serviu de base ao processo de inferência. O RPython não possui qualquer definição formal. Assim, com a definição das regras do sistema de tipos pretendíamos formalizar a sintaxe desta linguagem.

A motivação para este trabalho reside no desejo de certificar programas Python como correctos e seguros. Para tal, desejamos traduzir programas Python anotados em programas na linguagem HL, a qual é usada pelo gerador de obrigações de prova Why

[Fil03]. Ao contrário do HL, o Python não é tipificado explicitamente. Recorrendo à inferência de tipos pode-se anotar os programas Python com os tipos inferidos, o que ajudará no processo de tradução entre estas linguagens.

1.2 Trabalho relacionado

Existem alguns trabalhos relacionados com a inferência de tipos em Python. Uma abordagem sobre o tema é dada na tese de mestrado *Localized Type Inference of Atomic Types in Python* [Can05], onde se investiga a implementação de um algoritmo de inferência de tipos (atómicos) para o Python, preservando a semântica da linguagem. São também estudados os benefícios da adição de anotações de tipos à linguagem.

O Psyco [Rig04] é um compilador *just-in-time*(JIT) que emite código máquina *on the fly*, em vez de interpretar o programa Python passo a passo. Apesar de não proceder à inferência de tipos, o Psyco infere, em tempo de execução, algumas restrições para as variáveis, por exemplo, que uma determinada variável contém sempre um inteiro, ou um tuplo que possui na primeira posição um zero. Através destas restrições, é emitido código máquina eficiente.

O Starkiller [Sal00] é uma ferramenta que compila e infere tipos para Python, desenhado para gerar código máquina eficiente. A ferramenta analisa programas Python e transforma-os em programas C++ equivalentes. O Starkiller suporta toda a linguagem Python, excepto a inserção de código dinâmico (por exemplo, através da instrução `eval` ou `exec`).

A inferência estática de tipos em Python envolve restrições à linguagem, como já foi referido. John Aycock, no artigo *Aggressive Type Inference*[Ayc04], defende estas restrições, justificando que por o Python ser uma linguagem dinâmica isso não significa que o programador use código dinâmico frequentemente.

O Ruby é uma linguagem de alto-nível, tal como o Python, dinamicamente tipificada. No artigo *Static Type Inference for Ruby*[Fea09] é descrito um algoritmo para a

inferência estática de tipos nesta linguagem. O objectivo dos autores era adicionar uma disciplina de tipos à linguagem, que fornecesse aos programadores uma verificação adicional dos seus programas.

1.3 Resumo da dissertação

No segundo capítulo da dissertação introduzimos os conceitos importantes para o trabalho efectuado, relacionados com sistemas de tipos.

No terceiro capítulo descrevemos as características principais da linguagem Python, e quais as diferenças mais relevantes entre esta linguagem e o RPython.

No quarto capítulo descrevemos o sistema de tipos que desenvolvemos, apresentando os tipos e as regras para cada construtor.

No quinto capítulo descrevemos uma implementação do sistema de inferência de tipos, baseado no sistema de tipos definido no capítulo anterior. De seguida, ilustramos esta descrição com um exemplo prático.

Por fim, no último capítulo, apresentamos algumas conclusões finais e comentários de trabalho futuro.

Capítulo 2

Sistemas de Tipos

Um tipo corresponde a um conjunto de valores que possuem características comuns, que os distinguem dos outros valores. Os valores deste conjunto podem ser estruturas simples, como números inteiros, ou representar especificações de programas.

Suponhamos uma linguagem, vamos designá-la L, onde apenas podemos adicionar números inteiros (n) ou palavras (l):

$$e ::= n \quad | \, l \quad | \, e + e$$

Na linguagem L um valor só pode ter tipo `inteiro` ou `string`, uma vez que apenas podemos ter números inteiros ou palavras:

$$t ::= \quad int \quad | \, string$$

O sistema de tipos é um componente das linguagens tipificadas, isto é, das linguagens em que podem ser atribuídos tipos às variáveis e funções, e consequentemente aos construtores do programa. No âmbito da computação, o principal objectivo do uso de um sistema de tipos é prevenir a ocorrência de determinados erros durante a execução do programa. Para tal, um sistema de tipos define um conjunto de regras que associam

tipos aos construtores do programa. Cada construtor pode ter uma ou mais regras que o definem no sistema de tipos.

A associação de um tipo τ a um construtor `e` designa-se por atribuição de tipo, e representa-se por `e::`τ. Seja um contexto Γ um conjunto de atribuições de tipo a expressões. Dado um contexto Γ, um construtor `e` e um tipo τ, $\Gamma \vdash$ `e` $::\tau$ significa que considerando o contexto Γ é possível deduzir que o construtor `e` tem tipo τ. Assim, dizemos que um sistema de tipos é constituído por uma definição de um conjunto de tipos e por regras que permitem definir a atribuição de tipos a expressões, num dado contexto.

Suponhamos que na linguagem L a soma só podia ser efectuada entre dois inteiros ou duas `strings`. As regras que definem a operação soma no sistema de tipos são:

$$\frac{n_1 :: int \qquad n_2 :: int}{n_1 + n_2 :: int} \qquad \text{(SOMAI)}$$

$$\frac{l_1 :: string \qquad l_2 :: string}{l_1 + l_2 :: string} \qquad \text{(SOMAS)}$$

O sistema de tipos permite-nos saber se um programa é "bem comportado", isto é, se um programa respeita as regras definidas pelo sistema de tipos. Se num programa, em linguagem L, tivéssemos a soma de um inteiro com uma string, o programa não era "bem comportado", pois não respeitava nenhuma das regras para a operação soma, definidas no sistema de tipos. Desta forma, ao estarmos perante esta instrução seria lançado um *erro de conflito de tipos*.

As linguagens em que os tipos são parte da linguagem designam-se linguagens tipificadas explicitamente. Nestas linguagens, o sistema de tipos permite fazer a verificação de tipos e detectar erros de conflito de tipos num determinado programa.

A principal bibliografia consultada para a escrita deste capítulo foi B. Pierce [Pie02] e L. Cardelli e P. Wegner[CW85].

2.1 Verificação estática de tipos

As linguagens que garantem,em tempo de compilação,"bom comportamento" do programa, designam-se linguagens verificadas estaticamente.

A verificação estática de tipos, como analisa o programa em tempo de compilação, rejeita alguns programas que, na realidade, se comportam bem em tempo de execução. Consideremos o programa Programa 2.1.

```
1 def g(f,x,y):
2     if f(x)==f(y):
3         return f(x)
4     else:
5         return f(y)
```

Programa 2.1

Como em tempo de compilação não conhecemos os argumentos da função `g`, e consequentemente o seu tipo, não podemos garantir que não haja qualquer erro de conflito de tipos na definição da função. Desta forma, o programa é rejeitado.

O benefício mais óbvio da verificação estática de tipos é permitir a detecção precoce de alguns dos erros do programa. Os erros detectados precocemente podem ser corrigidos imediatamente. Além disso, a origem dos erros pode ser identificada com mais precisão durante a verificação de tipos do que em tempo de execução. No entanto, a verificação estática do programa reduz a flexibilidade do mesmo, uma vez que o sistema de tipos da linguagem é mais restritivo.

O uso de tipos tem também um papel importante na documentação do programa, pois facilita a leitura, o entendimento e a manutenção pelo programador, com vantagem sobre os comentários no programa, pois os tipos são verificados pelo compilador, enquanto os comentários podem estar desactualizados ou escritos de forma errada.

2.2 Verificação dinâmica de tipos

As linguagens não verificadas estaticamente podem garantir "bom comportamento" recorrendo a verificações, suficientemente detalhadas, em tempo de execução. Estas linguagens designam-se linguagens verificadas dinamicamente. O Python é um exemplo deste tipo de linguagens.

A verificação dinâmica de tipos não garante, em tempo de compilação, o uso correcto dos valores. Apenas em tempo de execução, caso o programa tente realizar operações que violam as regras do sistema de tipos, é lançado um erro.

O programa do exemplo anterior (Programa 2.1), ao ser verificado em tempo de execução, é aceite, pois como os argumentos já são conhecidos, e consequentemente os seus tipos, é possível concluir a não existência de erros de conflito de tipos.

A verificação dinâmica pode ter um baixo desempenho, uma vez que não é possível fazer optimizações do código, referentes à verificação de tipos, antes de o executar. No entanto, os programas são mais simples e expressivos uma vez que não temos que nos preocupar com as restrições impostas pelo sistema de tipos.

2.3 Inferência de tipos

Nas linguagens em que não se declara o tipo das variáveis ou funções, mas para as quais existe um sistema de tipos associado, os tipos poderão ser obtidos por um processo de inferência.

A inferência de tipos é o problema de encontrar um tipo para os construtores, dentro de um sistema de tipos. Este problema não é fácil de resolver, sendo muitas vezes um problema indecidível.

Um algoritmo de inferência de tipos não deve encontrar, apenas, um tipo possível para uma determinada expressão, mas sim todos os tipos possíveis para a expressão dentro

do sistema de tipos.

Em 1978, foi proposto por Robin Milner um sistema de inferência de tipos para a linguagem ML [Mil78], sendo este melhorado e estendido por Luís Damas em 1982 [DM82]. A ideia apresentada para este sistema de tipos tem sido aplicada noutras linguagens de programação.

2.4 Sistemas de tipos não polimórficos

O λ-calculus com tipos simples [Hin97], por vezes designado como sistema F_1, é o exemplo de uma linguagem com um sistema de tipos não polimórfico. Este sistema está na base das linguagens de programação funcionais, como o Haskell e o ML.

Neste sistema, a abstracção λx.M representa a função com argumento x e resultado M. Dado um conjunto infinito de variáveis *Var*, e sendo x $\in$ *Var* e M e N termos, temos a seguinte sintaxe para o conjunto de termos:

$M, N ::=$	termos
x	variáveis
$\mid(\lambda x.M)$	função
$\mid (MN)$	aplicação

Assumindo um conjunto de variáveis de tipo VT, e $a \in VT$, a sintaxe dos tipos é dada por:

$\alpha, \tau ::=$	tipos
a	variáveis de tipo
$\mid\alpha \rightarrow \tau$	tipos de funções

A correspondência de um tipo τ a uma variável x designa-se por atribuição de tipo, e representa-se por $x : \tau$. Seja um contexto Γ, um conjunto de atribuições de tipo a

variáveis distintas. Dado um contexto Γ, um termo M e um tipo τ, $\Gamma \vdash M :: \tau$ indica que considerando o contexto Γ é possível deduzir que o termo M tem tipo τ.

O sistema de tipos simples define-se pelas seguintes regras de dedução:

$$\frac{x :: \alpha \in \Gamma}{\Gamma \vdash x :: \alpha} \qquad \text{(Axioma)}$$

$$\frac{\Gamma, x :: \alpha \vdash M :: \beta}{\Gamma \vdash (\lambda x.M) :: (\alpha \to \beta)} \qquad \text{(Abstracção)}$$

$$\frac{\Gamma \vdash M :: \alpha \to \beta \qquad \Gamma \vdash N :: \alpha}{\Gamma \vdash MN :: \beta} \qquad \text{(Aplicação)}$$

Uma atribuição de tipo $\Gamma \vdash M : \alpha$ só pode ser obtida pela aplicação destas regras de inferência.

Podemos derivar, por exemplo, um tipo para o termo $\lambda x(\lambda y.x)$, usando as regras anteriores:

$\{x :: \alpha \to \alpha, y :: \beta\} \vdash x :: \alpha \to \alpha$

Abstracção $\downarrow$

$\{x :: \alpha \to \alpha\} \vdash (\lambda y.x) :: \beta \to \alpha \to \alpha$

Abstracção $\downarrow$

$\vdash \lambda x(\lambda y.x) :: (\alpha \to \alpha) \to \beta \to \alpha \to \alpha$

2.5 Sistemas de tipos polimórficos

Um sistema de tipos é polimórfico se permite que uma única função possa ser usada com vários tipos.

Existem vários tipos de polimorfismo [CW85]. Os principais são:

polimorfismo paramétrico permite que uma função seja verificada de forma genérica, usando variáveis de tipo no lugar dos tipos e, instanciando-as, quando necessário, com tipos específicos. Assim, uma única função pode ser aplicada a um conjunto de valores, sem qualquer relação entre si. Por exemplo, a função:

```
1 def f(x):
2     return [x]
```

Programa 2.2

tem tipo $\forall a.\ a \rightarrow list\ a$, onde a é a variável de tipo que pode ser substituída por qualquer tipo específico.

polimorfismo ad-hoc permite definir funções cujo comportamento vai depender do tipo dos argumentos. Suponhamos, por exemplo, uma função f que quando recebe um inteiro retorna um `float`, mas que quando recebe uma `string` retorna um booleano. O comportamento da função f depende do tipo do argumento que recebe.

As diferentes formas de polimorfismo não são exclusivas, elas podem ser misturadas na mesma linguagem. Por exemplo, o ML oferece uma forma restrita de polimorfismo paramétrico e polimorfismo ad-hoc das operações aritméticas simples; enquanto que o Java inclui polimorfismo ad-hoc simples.

2.6 Subtipos

Os subtipos exprimem a noção intuitiva da inclusão entre tipos, onde os tipos são vistos como colecções de valores. Um elemento de um determinado tipo pode ser considerado também elemento de um dos seus super-tipos, o que permite que um valor seja usado flexivelmente em diferentes contextos.

Numa relação de subtipificação dizemos que A é subtipo de B (A<:B) se qualquer termo de tipo A pode ser usado no contexto onde um termo de tipo B é esperado.

Uma forma simples de entender o subtipificação é ler A<:B como “todo o programa de tipo A é também programa de tipo B”.

Capítulo 3

Python

3.1 Características da linguagem

O Python é uma linguagem de programação de muito alto nível orientada a objectos. Esta linguagem possui uma sintaxe clara, que facilita a legibilidade do código e o desenvolvimento de programas.

O Python é dinamicamente tipificado, isto é, a verificação de tipos é realizada em tempo de execução. Esta linguagem permite que as variáveis assumam valores de diversos tipos, em diferentes lugares do programa. Quando uma atribuição é avaliada, é dado um tipo à variável do lado esquerdo da atribuição, dependendo do tipo das operações presentes do lado direito. Esta variável podia até já ter um tipo, mas no momento da atribuição ela adquire o novo tipo. Consideremos o seguinte código:

```
1 x=3
2 if True: print x
3 x=False
```

Programa 3.1

Quando a primeira atribuição é analisada é atribuído a `x` o tipo inteiro. Assim, quando é usado na instrução if, `x` tem tipo inteiro, pois não existiu mais nenhuma atribuição

que lhe alterasse o tipo. No entanto, com a segunda atribuição, o tipo de `x` é alterado, e este passa a ter tipo booleano.

O Python tem diversos tipos predefinidos:

Tipos numéricos que podem ser: inteiros, longos, vírgula flutuante e complexos.

Booleanos que é uma especialização do tipo inteiro. O verdadeiro é chamado `True` e é igual a 1, enquanto o falso é chamado `False` e é igual a zero.

None que representa o valor nulo.

Strings que podem ser um único caractere. Não existe diferença entre caracteres e `strings`.

Tuplos que são sequências de valores, que podem ser acedidos individualmente. Estes valores podem ter qualquer tipo, no entanto depois do tuplo estar criado não pode ser alterado – objecto imutável.

Listas que são um conjunto de valores, indexados pela posição dos valores na lista. Os itens da lista podem ser de qualquer tipo. Ao contrário dos tuplos, as listas podem ser modificadas depois de criadas – objectos mutáveis. Em qualquer altura, podemos adicionar ou remover elementos da lista.

Dicionários que podem ser vistos como um conjunto não ordenado de pares (chave :valor), onde as chaves são únicas. Ao contrário das listas, os dicionários são indexados por chaves. Estas, podem ser de qualquer tipo imutável: `strings` e números podem sempre ser chaves; os tuplos podem ser chaves se contém apenas `strings`, números ou tuplos; se um tuplo contém qualquer objecto mutável, directa ou indirectamente, ele não pode ser usado como chave. Tal como as listas são objectos mutáveis.

3.1.1 Criacção de objectos em Python

As classes são estruturas fundamentais para definir novos objectos. Podemos, por exemplo, definir tipos de dados específicos através delas. São definidas pelo construtor `class`.

As classes, como todos os outros objectos, possuem um nome, um conjunto de atributos e métodos. Os métodos são as funções definidas dentro da classe. O primeiro argumento de um método é especial, pois representa a própria classe onde este está a ser definido. Convenciona-se chamar este argumento de *self*. Os atributos são estruturas de dados onde se guarda informação.

No seguinte exemplo temos uma classe de nome `par`, que define um atributo (`atr`) e dois métodos (`fst` e `snd`).

```
1 class par():
2     atr =None
3     def __init__(self,z):
4         atr=z
5     def fst(self,x,y):
6         return x
7     def snd(self,x,y):
8         return y
```

O método __init__ é opcional e, se existe, é invocado quando uma classe é instanciada, ou seja, quando é criado um objecto (instância) de uma classe. Os métodos base de uma instância são definidos pela classe a partir da qual a instância é criada. No entanto, em qualquer altura, estes métodos podem ser alterados. Depois de criada uma instância, podemos aceder aos seus métodos usando a sintaxe `nome_da_classe.nome_do_método`.

Retomando o exemplo anterior, podemos instanciar um objecto `c`, e aceder aos seus métodos `fst` e `snd`:

```
c=par(3)
f=c.fst(1,2)
s=c.snd(2,3)
```

Com o exemplo torna-se mais claro o uso do argumento *self*: este representa a instância sobre a qual o método vai ser invocado.

3.1.2 Herança no Python

O Python suporta herança, isto é, permite derivar novas classes a partir de classes já existentes, denominadas neste contexto *classes-base*. As classes derivadas possuem acesso aos atributos e métodos das classes-base, e podem redefini-los conforme seja conveniente. A herança é uma forma simples de promover a reutilização de dados.

Vamos adicionar ao exemplo anterior, uma nova classe designada `hers`. Esta classe herda da classe `par`, definida anteriormente.

```
class hers(par):
    def __init__(self,x,y):
        self.x=x
        self.y=y
    def imp(self):
        return self.fst(self.x,self.y)
```

Usando a herança, podemos definir um método `imp` que acede ao método `fst` definido na classe `par`. Se não usássemos herança teríamos que voltar a definir o método `fst` para o podermos usar.

O Python suporta também herança múltipla, ou seja, permite que uma classe herde dados e métodos de várias classes diferentes. Vamos estender o exemplo com uma classe sem herança (`num`), e outra com herança múltipla (`herm`):

```
1  class num():
2      def __init_(self,n):
3          print n
4      def succ(self,n):
5          return n+1
6
7  class herm (par,num):
8      def __init__(self,x,y):
9          print x,y
10     def imp(self,x,y):
11         t=self.fst(x,y)
12         return self.succ(t)
```

Deste modo, a classe `herm` herdou os métodos das classes `par` e `num`, podendo usá-los sempre que necessitar. Caso existissem métodos com o mesmo nome na classe `par` e na classe `num`, a classe `par`, por ser a primeira na linha de definição da classe `herm`, tinha prioridade sobre a classe `num`.

3.1.3 Introspecção e reflexão no Python

A introspecção e reflexão são propriedades das linguagens orientadas a objectos que qualificam a existência de mecanismos para descobrir e alterar, em tempo de execução, informações estruturais sobre um programa e objectos existentes neste.

O Python possui tanto características introspectivas quanto reflexivas, uma vez que, permite obter em tempo de execução informações a respeito do tipo dos objectos, incluindo informações sobre a hierarquia de classes. Preserva também dados que descrevem a estrutura do programa que está a ser executado, permitindo que se estude como este está organizado, sem a necessidade de ler o seu código-fonte.

Em qualquer momento da execução podemos, por exemplo, determinar o tipo de um objecto, usando a função `type(objecto)`. Se usarmos a função `dir(objecto)`

sabemos todos os nomes dos atributos definidos no objecto. Podemos determinar quais as classes das quais uma outra herda métodos e atributos chamando a função `classe.__bases__`.

3.2 PyPy

O objectivo inicial do projecto PyPy foi escrever um interpretador de Python em Python, de forma a ter a linguagem descrita nela própria. Executar um interpretador por cima de outro conduz a uma execução lenta. Assim, foi decidido construir um conjunto de ferramentas que traduzem especificações de alto-nível dos interpretadores (máquinas virtuais) de Python, em *backends* de baixo nível como C/Posix. Tal como o interpretador o conjunto de ferramentas está escrito em Python [RPH05].

O PyPy é constituído por duas partes principais: o interpretador e o conjunto de ferramentas de tradução [RP06].

3.2.1 Interpretador

O interpretador implementa a linguagem Python completa e está escrito em RPython, um subconjunto do Python que está descrito na Secção 3.2.3. Este interpretador é constituído por:

- um compilador que traduz o código Python em bytecode;
- um avaliador de bytecode que avalia o bytecode gerado na fase anterior;
- um espaço básico de objectos onde os objectos Python são criados e manipulados. Este espaço de objectos permite criar um grafo de fluxo a partir de cada função.

O grafo de fluxo é a representação de uma abstracção da execução de um programa. Este nem sempre é sequencial, isto é, existem partes do programa que apenas são executadas se determinada condição se verificar. Desta forma, os nós do grafo representam

os blocos sequenciais do programa. Os arcos do grafo ligam dois blocos sequenciais caso uma dada condição se verifique.

A Figura 3.1 esquematiza a arquitectura do PyPy, ilustrando o comportamento do interpretador.

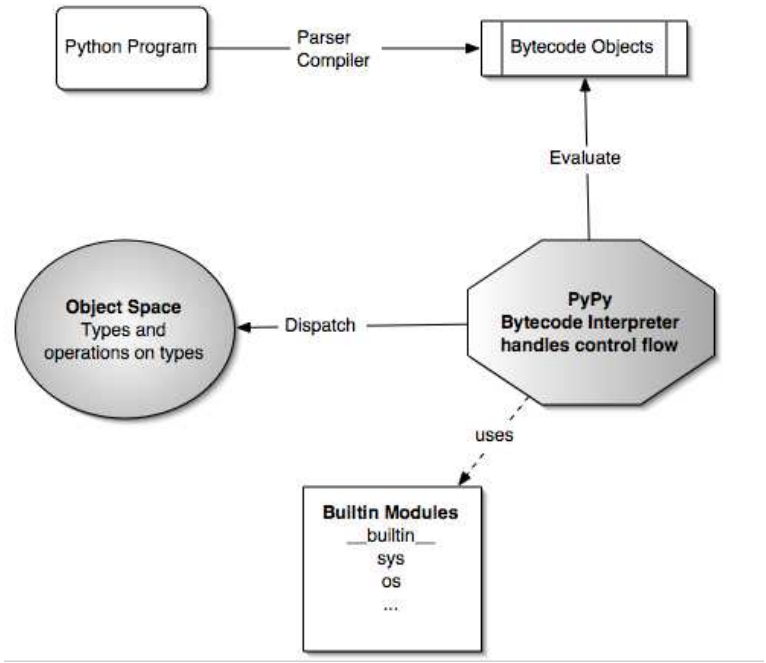

Figura 3.1: Arquitectura básica do PyPy [Ped]

3.2.2 Conjunto de ferramentas de tradução

O processo de tradução inicia-se com um programa RPython, que posteriormente é traduzido numa versão eficiente para uma ou mais plataformas. A Figura 3.2 esquematiza este processo.

Os passos do processo de tradução são:

- O tradutor converte o programa de entrada RPython em grafos de fluxo. Esta conversão é realizada pelo interpretador, o qual também faz parte do conjunto

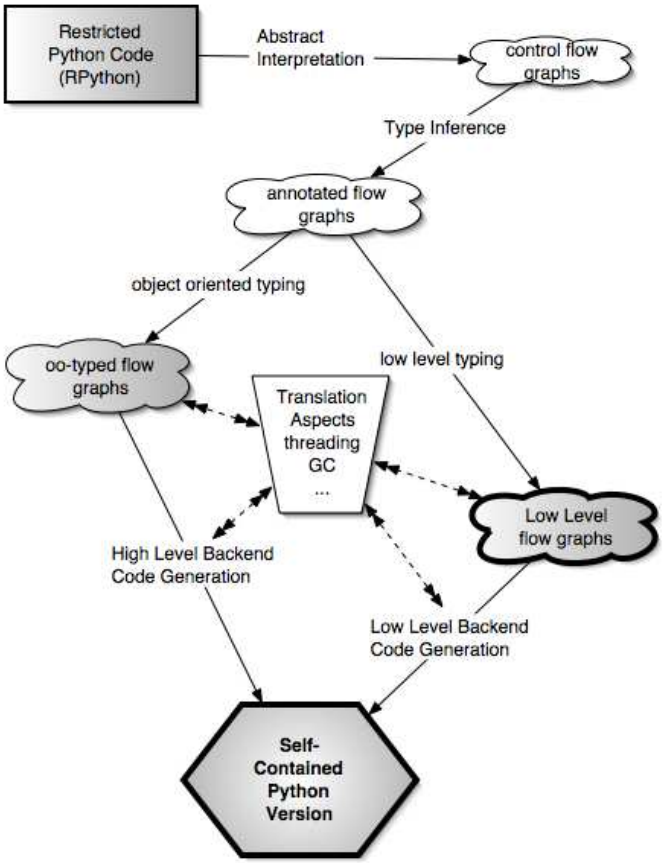

Figura 3.2: Processo tradução PyPy [Ped]

de ferramentas de tradução. Cada passo da tradução baseia-se na transformação destes grafos de fluxo.

- Após os grafos de fluxo estarem construídos procede-se à inferência de tipos, anotando os nós dos grafos com os tipos inferidos. No fim da análise, o programa é representado por uma floresta de grafos de fluxo anotados.

- O passo de anotação é seguido pela conversão dos grafos de alto-nível em grafos

de baixo-nível, ou seja, à substituição dos tipos e operações de alto-nível, para as correspondentes em baixo-nível. Até ao momento, existem dois conversores, um especializado em baixo nível semelhante ao C, e outro especializado em baixo-nível orientado a objectos, como o CLI.

- Após a conversão são aplicadas optimizações. Finalmente, o grafo fica preparado para a geração do código na plataforma pretendida.

3.2.3 RPython

A linguagem RPython é um sub-produto do PyPy. Esta foi uma linguagem conveniente no desenvolvimento do projecto.

O RPython é um subconjunto do Python suficientemente estático para permitir inferência de tipos. Por suficientemente estático queremos dizer que este subconjunto apenas tem as características do Python que nos permitem inferir tipos em tempo de compilação.

Ao contrário do que acontece em Python, em RPython as variáveis têm que ter tipo consistente, isto é, apenas podem conter valores de um único tipo. Os tipos predefinidos suportados em RPython são:

- SomeInteger;
- SomeFloat;
- SomeBool;
- SomeChar e SomeString: embora o Python não distinga entre `strings` e caracteres, o RPython usa tipos diferentes para eles: durante a inferência de tipos, as `strings` com exactamente tamanho um têm tipo SomeChar.
- SomeTuple: representa os tuplos, os quais são usados quer em Python quer em RPython para agrupar conjuntos de objectos que podem ter diferentes tipos.

- SomeList: é usado para guardar sequências de itens mutáveis, os quais, ao contrário do que acontece em Python, têm que ser todos o mesmo tipo.
- SomeDict: que é equivalente aos dicionários em Python. No entanto, em RPython têm que ser homogéneos (as chaves têm que ter o mesmo tipo, assim como os valores; mas as chaves não têm que ter o mesmo tipo dos valores).

O prefixo Some presente nos tipos é apenas uma convenção usada internamente pelo anotador do PyPy.

Além dos tipos predefinidos podem, tal como em Python, ser definidos tipos específicos, através das classes. No entanto, o RPython não permite a mudança dinâmica das classes, adicionando ou removendo métodos. Cada objecto pertence a uma classe e cada classe tem um conjunto fixo de métodos. Não é permitido também o uso de métodos especiais (__*__), assim como o uso das capacidades reflexivas do Python. Apenas é permitida a herança simples. Contudo, o RPython suporta definições que permitem que métodos sejam partilhados por várias classes (definições "mixin").

Embora todas esta restrições pareçam ser substanciais para uma linguagem dinâmica como o Python, continua a ser possível usar características de alto-nível, como por exemplo a herança (simples).

Capítulo 4

Sistema de tipos para o Python

Um sistema de tipos permite prevenir a ocorrência de determinados erros durante a execução do programa. Para tal, define-se um conjunto de regras que associam tipos aos construtores de um programa.

A linguagem que usamos neste sistema de tipos é um subconjunto da linguagem RPython, apresentado na Secção (3.2.3). No entanto, existem algumas diferenças entre as duas linguagens:

- enquanto que o RPython distingue entre caracteres e `strings`, nós não o fazemos;
- em RPython as chaves dos dicionários têm que ter o mesmo tipo, nós apenas exigimos que elas sejam *hashable*, isto é, que sejam números inteiros, `floats`, `strings` ou tuplos.

4.1 Sistema de tipos sem polimorfismo

4.1.1 Sintaxe

A sintaxe da linguagem é a seguinte:

$$\begin{aligned}\texttt{e}, \bar{\texttt{e}} ::= \quad & \texttt{n} \mid \texttt{l} \mid \texttt{x} \mid (\texttt{e}_1, ..., \texttt{e}_n) \mid [\texttt{e}_1, ..., \texttt{e}_n] \mid \{\bar{\texttt{e}}_1 : \texttt{e}_1, ..., \bar{\texttt{e}}_n : \texttt{e}_n\} \mid \texttt{x} = \texttt{e} \mid \texttt{e op e} \\ & \mid \texttt{e opc e} \mid \texttt{e opb e} \mid \texttt{opu e} \mid \texttt{if e : e else e} \mid \texttt{e[n]} \mid \texttt{return} \mid \texttt{return e} \\ & \mid \texttt{while e : e else e} \mid \texttt{def f } (\texttt{e}_1, ..., \texttt{e}_n) \texttt{ : e} \mid \texttt{f } (\texttt{e}_1, ..., \texttt{e}_n)\end{aligned}$$

onde,

$\texttt{n} \in \{\text{int, long, float}\}$

$\texttt{l} \in$ constantes

$\texttt{x} \in$ nomes de variáveis

$\texttt{f} \in$ nomes de funções

$$\texttt{op} ::= \ + \mid - \mid * \mid << \mid >> \mid | \mid \hat{} \mid \& \mid / \mid \% \mid ** \mid //$$

$$\texttt{opc} ::= \ == \mid != \mid < \mid \leq \mid > \mid \geq \mid \texttt{ is } \mid \texttt{ not is } \mid \texttt{ in } \mid \texttt{ not in}$$

$$\texttt{opb} ::= \ \texttt{and} \mid \texttt{or}$$

$$\texttt{opu} ::= \ \texttt{not} \mid \sim \mid + \mid -$$

4.1.2 Tipos

O conjunto de tipos deste sistema define-se pela seguinte gramática, onde `TVar` representa o conjunto das variáveis de tipo:

$$\begin{aligned}\tau, \alpha ::= \quad & \texttt{eInt} \mid \texttt{eFloat} \mid \texttt{eLong} \mid \texttt{eString} \mid \texttt{eBool} \mid \texttt{eNone} \mid \texttt{eTop} \mid \sigma \in \texttt{TVar} \\ & \mid \texttt{eTuple}(\tau_1, ..., \tau_n) \mid \texttt{eList}(\tau) \mid \texttt{eDict}(\tau) \mid \texttt{eArrow}([\tau_1, ..., \tau_n], \alpha)\end{aligned}$$

Para associarmos um tipo τ a um valor `e` fazemos uma atribuição de tipo da forma `e::`τ. Vamos, de seguida, mostrar exemplos que ilustram a utilização dos tipos:

```
 1 ::  eInt

1.0 ::  eFloat

1L ::  eLong

‘‘a’’::  eString

(1,‘‘a’’,1.0) ::  eTuple(eInt, eString,eFloat)

[1,2]::  eList(eInt)

{‘‘u’’:1.0,‘‘d’’:2.0} ::  eDict(eFloat)

{} ::  eDict(eNone)

def f(1):  return 1.0 ::  eArrow([eInt],eFloat)

def f(x,y):  return x ::  eArrow([σ1, σ2],σ1)
```

4.1.3 Subtipos

Vamos considerar a noção de subtipos, já descrita na Secção 2.6. As regras que exprimem a relação de subtipificação para os tipos definidos na Secção 4.1.2 são:

$$\tau <: \tau \qquad \text{(Ref)}$$

$$\texttt{eNone} <: \tau \qquad \text{(Sen)}$$

$$\texttt{eTop} <: \tau \qquad \text{(Sdv)}$$

$$\texttt{eInt} <: \texttt{eLong} \qquad \text{(Sil)}$$

$$\texttt{eInt} <: \texttt{eFloat} \qquad \text{(Sif)}$$

$$\texttt{eLong} <: \texttt{eFloat} \qquad \text{(Slf)}$$

$$\frac{\tau_{\mathtt{i}} <: \alpha_{\mathtt{i}} \qquad 1 \leq \mathtt{i} \leq \mathtt{n}}{\mathtt{eTuple}(\tau_1, ..., \tau_{\mathtt{n}}) <: \mathtt{eTuple}(\alpha_1, ..., \alpha_{\mathtt{n}})} \tag{ST}$$

$$\frac{\tau <: \alpha}{\mathtt{eList}(\tau) <: \mathtt{eList}(\alpha)} \tag{SL}$$

$$\frac{\tau <: \alpha}{\mathtt{eDict}(\tau) <: \mathtt{eDict}(\alpha)} \tag{SD}$$

$$\frac{\mathtt{unify}(\xi[\sigma_1], \xi[\sigma_2])}{\sigma 1 <: \sigma_2} \tag{SV}$$

$$\frac{\mathtt{unify}(\xi[\sigma], \tau)}{\sigma <: \tau} \tag{SV1}$$

$$\frac{\mathtt{unify}(\xi[\sigma], \tau)}{\tau <: \sigma} \tag{SV2}$$

$$\frac{\alpha_{\mathtt{i}} <: \tau_{\mathtt{i}} \qquad 1 \leq \mathtt{i} \leq \mathtt{n} \qquad \tau <: \alpha}{\mathtt{eArrow}([\tau_0, ..., \tau_{\mathtt{i}}], \tau) <: \mathtt{eArrow}([\alpha_1, ..., \alpha_{\mathtt{i}}], \alpha)} \tag{SF}$$

A regra de subtipificação para os tuplos (ST) diz-nos que um tuplo A é subtipo de um tuplo B se ambos têm o mesmo número de elementos, e se o i-ésimo elemento do tuplo A é subtipo do i-ésimo elemento do tuplo B.

Na regra (SV*) é usada a função `unify`, a qual faz a unificação de dois tipos passados como argumento. A unificação é o processo que permite, através de substituições, tornar os dois tipos passados como argumento idênticos. A substituição de um tipo τ por uma variável de tipo σ representa-se por $[\sigma \mapsto \tau]$. Seja ξ um dicionário onde estas substituições são efectuadas. As variáveis são inicializadas com o seu próprio valor, isto é, caso tenhamos uma variável de tipo σ que ainda não tenha sofrido qualquer substituição: $\xi = \{\sigma : \sigma\}$. Quando procedemos, por exemplo, a uma substituição

$[\sigma \mapsto \tau]$ temos que $\xi = \{\sigma : \tau\}$. Todas as ocorrências de σ em ξ têm que ser substituídas.

Consideremos dois tipos τ_1 e τ_2, o algoritmo $\texttt{unify}(\tau_1, \tau_2)$ é definido por:

$$\texttt{unify}(\sigma, \tau) = \begin{cases} [\sigma \mapsto \tau] & \texttt{se } \tau \neq \sigma \\ \texttt{id} & \texttt{se } \tau = \sigma \end{cases}$$

$$\texttt{unify}(\tau, \sigma) = \texttt{unify}(\sigma, \tau)$$

$$\texttt{unify}(\tau_1, \tau_2) = \tau_1 <: \tau_2$$

Suponhamos que queríamos verificar $\sigma <: \texttt{eInt}$, onde σ era uma variável de tipo. De acordo com as regras anteriores, seria chamada a função `unify` com os argumentos σ e `eInt`. Nesta chamada, a variável de tipo σ seria substituída por `eInt` em ξ.

Suponhamos agora que queríamos verificar $\texttt{eArrow}([\sigma_1, \sigma_2], \sigma_1) <: \texttt{eArrow}([\texttt{eInt}, \texttt{eFloat}], \texttt{eInt})$. Pelas regra Sf tínhamos que verificar $\sigma_1 <: \texttt{eInt}, \sigma_2 <: \texttt{eFloat}, \sigma_1 <: \texttt{eInt}$. Ao verificarmos $\sigma_1 <: \texttt{eInt}$ a variável de tipo σ_1 seria substituída por `eInt`, sendo o $\xi = \{\sigma_1 : \sigma_1, \sigma_2 : \sigma_2\}$ alterado para $\xi = \{\sigma_1 : \texttt{eInt}, \sigma_2 : \sigma_2\}$. A seguir quando verificarmos $\sigma_2 <: \texttt{eFloat}$, a variável de tipo σ_2 seria substituída pelo tipo `eFloat`, logo $\xi = \{\sigma_1 : \texttt{eInt}, \sigma_2 : \texttt{eFloat}\}$. Quando fizéssemos a última verificação ($\sigma_1 <: \texttt{eInt}$), já seria chamado o $\texttt{unify}(\texttt{eInt}, \texttt{eInt})$, uma vez que $\xi[\sigma_1] = \texttt{eInt}$, o que corresponderia a verificar $\texttt{eInt} <: \texttt{eInt}$.

A regra para as funções (Sf) pode causar mais confusão, uma vez que a relação dos tipos dos argumentos está em sentido contrário ao da relação entre os tipos das funções. Uma função M de tipo A $\rightarrow$ B aceita argumentos de tipo A, logo também aceita argumentos de tipo A', desde que A' seja subtipo de A. A mesma função M retorna argumentos de tipo B, logo retorna também argumentos de tipo B', desde que B' seja super-tipo de B. Assim, qualquer função M de tipo A $\rightarrow$ B é também de tipo A' $\rightarrow$ B', ou seja, A $\rightarrow$ B é subtipo de A' $\rightarrow$ B'.

Normalmente, as relações de subtipificação são transitivas. No nosso sistema de tipos esta relação não é necessária e, por isso, não é descrita a regra que exprimiria esta relação.

4.1.4 Regras do sistema de tipos

Ao conjunto das atribuições de tipo a variáveis ou funções, distintas, chamamos *contexto*, e representamos por Γ. A definição deste conjunto, onde $t_i \in x, f$, é a seguinte:

$$\Gamma ::= \{\mathtt{t_0} :: \tau_0, ..., \mathtt{t_n} :: \tau_n\}$$

Consideremos um subconjunto do contexto que apenas contém as atribuições de tipo correspondentes a função:

$$\Gamma_\mathtt{f} ::= \{\mathtt{t_i} :: \tau_\mathtt{i} \mid \tau_\mathtt{i} \ \mathtt{is\ eArrow(a, b)}\}$$

Vamos, de seguida, definir as regras do sistema de tipos, descrevendo algumas com mais pormenor:

$$\Gamma \vdash \mathtt{n} :: \mathtt{eInt} \qquad \text{(NumI)}$$

$$\Gamma \vdash \mathtt{n} :: \mathtt{eFloat} \qquad \text{(NumF)}$$

$$\Gamma \vdash \mathtt{n} :: \mathtt{eLong} \qquad \text{(NumL)}$$

$$\Gamma \vdash \mathtt{l} :: \mathtt{eString} \qquad \text{(Str)}$$

$$\Gamma \vdash \mathtt{True} :: \mathtt{eBool} \qquad \text{(BoolT)}$$

$$\Gamma \vdash \mathtt{False} :: \mathtt{eBool} \qquad \text{(BoolF)}$$

$$\Gamma \vdash \mathtt{x} :: \tau,\ \mathtt{se}\ (\mathtt{x} :: \tau) \in \Gamma \qquad \text{(Var)}$$

$$\frac{\Gamma \vdash \mathtt{e_i} :: \tau_\mathtt{i} \qquad 1 \leq \mathtt{i} \leq \mathtt{n}}{\Gamma \vdash (\mathtt{e_1}, ..., \mathtt{e_n}) :: \mathtt{eTuple}\ ([\tau_1, ..., \tau_\mathtt{n}])} \qquad \text{(Tuplo)}$$

$$\frac{\Gamma \vdash \mathtt{e_i} :: \tau \qquad 1 \leq \mathtt{i} \leq \mathtt{n}}{\Gamma \vdash [\mathtt{e_1}, ..., \mathtt{e_n}] :: \mathtt{eList}\ (\tau)} \qquad \text{(Lst)}$$

(Var) Uma variável `x` tem tipo τ, se a atribuição do tipo τ à variável `x` existe em Γ. Se a variável `x` não está presente em Γ, ou seja, se ainda não tem tipo atribuído, atribuímos-lhe o tipo σ.

(Tuplo) Um tuplo tem tipo $\mathtt{eTuple}([\tau_1, ..., \tau_\mathtt{n}])$, onde τ_i é o tipo do elemento na i-ésima posição do tuplo. Por exemplo, $(\mathtt{1},\ \text{“}\mathtt{a}\text{”},\ \mathtt{2.0}) :: \mathtt{eTuple}(\mathtt{eInt}, \mathtt{eString}, \mathtt{eFloat})$.

(Lst) As listas são objectos homogéneos, isto é, os seus elementos têm todos o mesmo tipo(τ). Assim, uma lista tem tipo $\mathtt{eList}(\tau)$. Por exemplo, $[\mathtt{1}, \mathtt{2}, \mathtt{3}] :: \mathtt{eList}(\mathtt{eInt})$.

$$\{\} :: \mathtt{eDict}(\mathtt{eNone}) \qquad \text{(DicV)}$$

$$\frac{\Gamma \vdash \bar{\mathtt{e}}_\mathtt{i} :: \alpha_\mathtt{i} \quad \mathtt{hashable}(\alpha_\mathtt{i}) \quad \Gamma \vdash \mathtt{e_i} :: \tau \qquad 1 \leq \mathtt{i} \leq \mathtt{n}}{\{\bar{\mathtt{e}}_1 : \mathtt{e_1}, ..., \bar{\mathtt{e}}_\mathtt{n} : \mathtt{e_n}\} :: \mathtt{eDict}\ (\tau)} \qquad \text{(Dic)}$$

(Dic*) Um dicionário pode estar vazio e nesse caso o seu tipo é `eDict(eNone)`. Quando um dicionário é constituído por pares (chave:valor), temos que garantir que o tipo da chave é `hashable` e inferir o tipo do valor. A função `hashable` define-se do seguinte modo:

$$\texttt{hashable}(\alpha) = \begin{cases} \texttt{True} & \texttt{se } \alpha = \{\texttt{eInt}, \texttt{eFloat}, \texttt{eLong}, \texttt{eString}, \texttt{eTuple}\} \\ \texttt{False} & \texttt{caso contrário} \end{cases}$$

Em todos os pares (chave:valor) o valor tem que ter o mesmo tipo τ. Por exemplo, $\{\text{“}\texttt{a}\text{”} : \texttt{1}, \text{“}\texttt{b}\text{”} : \texttt{2}\} :: \texttt{eDict}(\texttt{eInt})$, pois $\texttt{1} :: \texttt{eInt}$, $\texttt{2} :: \texttt{eInt}$ e “a” e “b” são `hashable`.

$$\frac{\Gamma \vdash \texttt{e} :: \tau_1 \qquad \Gamma \vdash \texttt{x} :: \tau_2 \qquad \tau_1 <: \tau_2}{\Gamma \vdash \texttt{x} = \texttt{e} \ :: \ \texttt{eNone}} \quad (\textsc{Atr}1)$$

$$\frac{\Gamma \vdash \texttt{e} :: \tau_1 \qquad \Gamma \vdash \texttt{x} :: \tau_2 \qquad \tau_2 <: \tau_1}{\Gamma \vdash \texttt{x} = \texttt{e} \ :: \ \texttt{eNone}} \quad (\textsc{Atr}2)$$

(Atr*) Uma atribuição de um valor `e` a uma variável `x` tem sempre tipo `eNone`, desde que esteja correctamente definida. Para que tal aconteça temos que inferir o tipo de `e` (τ_1) e o tipo de `x` (τ_2), e estes têm que ser subtipo um do outro.

Suponhamos que $\Gamma = \{\texttt{x} :: \texttt{eInt}\}$, então a atribuição $\texttt{x} = \texttt{1}$ tem tipo `eNone`. No entanto, a atribuição $\texttt{x} =$ “ola” lançará um erro porque existe um conflito de tipos. Suponhamos que $\Gamma = \{\texttt{x} :: \sigma\}$, então ao verificarmos $\texttt{TVar}(\texttt{x}) <: \texttt{eInt}$, será chamada a função $\texttt{unify}(\sigma, \texttt{eInt})$, sendo σ substituído pelo tipo `eInt` em ξ.

$$\frac{\Gamma \vdash \texttt{e}_1 :: \tau_1 \qquad \Gamma \vdash \texttt{e}_2 :: \tau_2 \qquad \tau_1 <: \tau_2}{\Gamma \vdash \texttt{e}_1 \ + \ \texttt{e}_2 \ :: \ \tau_2} \quad (\textsc{Opb}1)$$

$$\frac{\Gamma \vdash \texttt{e}_1 :: \tau_1 \qquad \Gamma \vdash \texttt{e}_2 :: \tau_2 \qquad \tau_2 <: \tau_1}{\Gamma \vdash \texttt{e}_1 \ + \ \texttt{e}_2 \ :: \ \tau_1} \quad (\textsc{Opb}2)$$

$$\tau_1, \tau_2 \ \in \ \{\texttt{eInt}, \ \texttt{eFloat}, \ \texttt{eLong}, \ \texttt{eString}, \ \texttt{eTuple}(\alpha), \ \texttt{eList}(\alpha)\}$$

$$\frac{\Gamma \vdash \mathtt{e}_1 :: \alpha \qquad \Gamma \vdash \mathtt{n} :: \mathtt{eInt}}{\Gamma \vdash \mathtt{e}_1 * \mathtt{n} \; :: \alpha} \tag{OPB3}$$

$\alpha \in \{\mathtt{eList}, \mathtt{eTuple}, \mathtt{eString}\}$

$$\frac{\Gamma \vdash \mathtt{e}_1 :: \tau_1 \qquad \Gamma \vdash \mathtt{e}_2 :: \tau_2 \qquad \tau_1 <: \tau_2}{\Gamma \vdash \mathtt{e}_1 \; \mathtt{op} \; \mathtt{e}_2 \; :: \tau_2} \tag{OPB4}$$

$$\frac{\Gamma \vdash \mathtt{e}_1 :: \tau_1 \qquad \Gamma \vdash \mathtt{e}_2 :: \tau_2 \qquad \tau_2 <: \tau_1}{\Gamma \vdash \mathtt{e}_1 \; \mathtt{op} \; \mathtt{e}_2 \; :: \tau_1} \tag{OPB5}$$

$\tau_\mathtt{i} \in \{\mathtt{eInt}, \mathtt{eLong}\} \quad \mathtt{op} \in \{\ll, \gg, |, \; \hat{}, \&\}$

$$\frac{\Gamma \vdash \mathtt{e}_1 :: \tau_1 \qquad \Gamma \vdash \mathtt{e}_2 :: \tau_2 \qquad \tau_1 <: \tau_2}{\Gamma \vdash \mathtt{e}_1 \; \mathtt{op} \; \mathtt{e}_2 \; :: \tau_2} \tag{OPB6}$$

$$\frac{\Gamma \vdash \mathtt{e}_1 :: \tau_1 \qquad \Gamma \vdash \mathtt{e}_2 :: \tau_2 \qquad \tau_2 <: \tau_1}{\Gamma \vdash \mathtt{e}_1 \; \mathtt{op} \; \mathtt{e}_2 \; :: \tau_1} \tag{OPB7}$$

$\tau_\mathtt{i} \in \{\mathtt{eInt}, \mathtt{eFloat}, \mathtt{eLong}\} \quad \mathtt{op} \in \{-, /, \%, *, **, //\}$

(OPB*) Na generalidade, as operações binárias são possíveis entre objectos com tipos que são subtipo um do outro. A operação tem o tipo mais geral desses dois tipos. Por exemplo, $1 + 1.0 \; :: \; \mathtt{eFloat}$, pois $\mathtt{eInt} <: \mathtt{eFloat}$.

$$\frac{\Gamma \vdash \mathtt{e}_1 :: \tau_1 \qquad \Gamma \vdash \mathtt{e}_2 :: \tau_2 \qquad \tau_1 <: \tau_2}{\Gamma \vdash \mathtt{e}_1 \; \mathtt{opc} \; \mathtt{e}_2 \; :: \mathtt{eBool}} \tag{OPC1}$$

$$\frac{\Gamma \vdash \mathtt{e}_1 :: \tau_1 \qquad \Gamma \vdash \mathtt{e}_2 :: \tau_2 \qquad \tau_2 <: \tau_1}{\Gamma \vdash \mathtt{e}_1 \; \mathtt{opc} \; \mathtt{e}_2 \; :: \mathtt{eBool}} \tag{OPC2}$$

(OPC*) Tal como nas operações binárias, os tipos dos operadores têm que ser subtipo um do outro. A operação tem sempre tipo eBool. Por exemplo, $1 < 2 :: \texttt{eBool}$.

$$\frac{\Gamma \vdash \ \texttt{e}_1 :: \texttt{eBool} \qquad \Gamma \vdash \texttt{e}_2 :: \texttt{eBool}}{\Gamma \vdash \texttt{e}_1 \ \texttt{opb} \ \texttt{e}_2 :: \texttt{eBool}} \qquad \text{(OPBOOL)}$$

(OPBOOL) Estas operações têm tipo eBool, assim como ambos os argumentos. Por exemplo $(2 < 3)\texttt{and}(4 > 1) :: \texttt{eBool}$, $2 < 3 :: \texttt{eBool}$, $4 < 1 :: \texttt{eBool}$.

$$\frac{\Gamma \vdash \ \texttt{e} :: \alpha}{\Gamma \vdash \texttt{not e} :: \texttt{eBool}} \qquad \text{(OPU1)}$$

$$\frac{\Gamma \vdash \ \texttt{e} :: \texttt{eInt}}{\Gamma \vdash \sim \texttt{e} :: \texttt{eInt}} \qquad \text{(OPU2)}$$

$$\frac{\Gamma \vdash \ \texttt{e} :: \tau}{\Gamma \vdash \texttt{op e} :: \tau} \qquad \text{(OPU3)}$$

$$\texttt{op} \in \{+, -\}, \qquad \tau \in \{\texttt{eInt}, \texttt{eFloat}, \texttt{eLong}\}$$

$$\frac{\Gamma \vdash \ \texttt{e} :: \texttt{eList}(\tau)}{\Gamma \vdash \texttt{e[i]} \ :: \tau} \qquad \text{(ALST1)}$$

$$\frac{\Gamma \vdash \ \texttt{e} :: \texttt{eList}(\tau)}{\Gamma \vdash \texttt{e[n : m]} \ :: \texttt{eList}(\tau)} \qquad \text{(ALST2)}$$

$$\frac{\Gamma \vdash \ \texttt{e} :: \texttt{eList}(\tau)}{\Gamma \vdash \texttt{e[n :]} \ :: \texttt{eList}(\tau)} \qquad \text{(ALST3)}$$

$$\frac{\Gamma \vdash \texttt{e} :: \texttt{eList}(\tau)}{\Gamma \vdash \texttt{e}[:\texttt{m}] \ :: \texttt{eList}(\tau)} \qquad \text{(ALST4)}$$

(ALST*) O tipo de um acesso a uma lista depende de se o acesso é feito apenas a um elemento da lista ou a uma parte dela. No primeiro caso, o acesso tem o tipo do elemento; no segundo, o resultado é uma lista, que tem o tipo da lista inicial. Por exemplo, seja $\texttt{l} :: \texttt{eList}(\texttt{eInt})$, $\texttt{l}[1] :: \texttt{eInt}$ e $\texttt{l}[:2] :: \texttt{eList}(\texttt{eInt})$.

$$\frac{\Gamma \vdash \texttt{e} :: \texttt{eDict}(\tau)}{\Gamma \vdash \texttt{e}[\texttt{i}] :: \tau} \qquad \text{(ADIC1)}$$

$$\frac{\Gamma \vdash \texttt{e} :: \texttt{eDict}(\texttt{eNone})}{\Gamma \vdash \texttt{e}[\texttt{i}] :: \texttt{eTop}} \qquad \text{(ADIC2)}$$

(ADIC*) Para aceder a um dicionário temos que verificar se ele está vazio. Caso esteja, o tipo do acesso é `eTop`, pois quaisquer valores podem ser adicionados ao dicionário. Caso contrário, o tipo do acesso é o tipo do dicionário. Suponhamos $\texttt{d} :: \texttt{eDict}(\texttt{eNone})$ e $\texttt{d}' :: \texttt{eDict}(\texttt{eInt})$, temos que $\texttt{d}[0] :: \texttt{eTop}$ e $\texttt{d}'[\text{“a”}] :: \texttt{eInt}$.

$$\Gamma \vdash \texttt{return} \ :: \texttt{eNone} \qquad \text{(RETURN1)}$$

$$\frac{\Gamma \ \vdash \ \texttt{e} :: \tau}{\Gamma \vdash \texttt{return e} \ :: \ \tau} \qquad \text{(RETURN2)}$$

(RETURN*) No caso em que nenhum valor é retornado o tipo da instrução é `eNone`. Caso contrário, a instrução tem o tipo do objecto que é retornado. Por exemplo, $\texttt{return 1} :: \texttt{eInt}$.

$$\frac{\Gamma \vdash \ \mathtt{e_0} :: \mathtt{eBool} \qquad \Gamma \vdash \mathtt{e_1} :: \tau \qquad \Gamma \vdash \mathtt{e_2} :: \alpha \qquad \tau <: \alpha}{\Gamma \vdash \mathtt{if\ e_0 :\ e_1\ else\ e_2\ } :: \alpha} \text{ (COND1)}$$

$$\frac{\Gamma \vdash \ \mathtt{e_0} :: \mathtt{eBool} \qquad \Gamma \vdash \mathtt{e_1} :: \tau \qquad \Gamma \vdash \mathtt{e_2} :: \alpha \qquad \alpha <: \tau}{\Gamma \vdash \mathtt{if\ e_0 :\ e_1\ else\ e_2\ } :: \tau} \text{ (COND2)}$$

(COND*) Num condicional o teste tem sempre tipo `eBool`. Os tipos dos dois ramos do condicional têm que ser subtipo um do outro. O tipo do condicional é o tipo mais geral destes dois. Por exemplo, `if` $2 < 3$ `: return 1.0 else : return 2` $::$ `eFloat`, pois $2 < 3 ::$ `eBool`, `return 1.0` $::$ `eFloat`, `return 2` $::$ `eInt` e `eInt` $<:$ `eFloat`.

$$\frac{\Gamma \vdash \mathtt{e_0} :: \mathtt{eBool} \qquad \Gamma \vdash \mathtt{e_1} :: \tau \qquad \Gamma \vdash \mathtt{e_2} :: \alpha \qquad \tau \ <: \alpha}{\Gamma \vdash \mathtt{while\ e_0 :\ e_1\ else\ e_2\ } :: \alpha} \text{ (WHILE1)}$$

$$\frac{\Gamma \vdash \mathtt{e_0} :: \mathtt{eBool} \qquad \Gamma \vdash \mathtt{e_1} :: \tau \qquad \Gamma \vdash \mathtt{e_2} :: \alpha \qquad \alpha \ <: \tau}{\Gamma \vdash \mathtt{while\ e_0 :\ e_1\ else\ e_2\ } :: \tau} \text{ (WHILE2)}$$

(WHILE*) A regra para a instrução `while` é semelhante ao condicional.

$$\frac{\bar{\Gamma} = \{\mathtt{x_i} :: \tau_\mathtt{i}\} \qquad 1 \leq \mathtt{i} \leq \mathtt{n} \qquad \bar{\Gamma} \cup \Gamma_\mathtt{f} \vdash \mathtt{e} :: \alpha}{\Gamma'' \vdash \mathtt{def\ f\ (x_1, ..., x_n) : e\ } :: \ \mathtt{eArrow}([\tau_1, ..., \tau_\mathtt{n}], \alpha)} \text{ (DEFFUNC)}$$

$\Gamma'' = \Gamma \cup f :: eArrow([\tau_1, ..., \tau_n], \alpha)$

$$\frac{\Gamma \vdash \mathtt{f} :: \mathtt{eArrow}([\tau_1, ..., \tau_\mathtt{n}], \alpha) \qquad \Gamma \vdash \bar{\mathtt{e}}_\mathtt{i} :: \alpha_\mathtt{i} \qquad \alpha_\mathtt{i} \ <: \ \tau_\mathtt{i} \qquad 1 \leq \mathtt{i} \leq \mathtt{n}}{\Gamma \vdash \mathtt{f}(\bar{\mathtt{e}_1}, ..., \bar{\mathtt{e}_n}) \ :: \ \alpha} \text{ (APLICAÇÃO)}$$

(DefFunc) As funções têm sempre tipo $\texttt{eArrow}([\tau_1, ..., \tau_2], \alpha)$, onde τ_i é o tipo do i-ésimo argumento da função e α é o tipo do corpo da mesma. O tipo dos argumentos é inferido num novo contexto $\bar{\Gamma}$. O corpo da função é inferido num contexto resultante da união do contexto $\bar{\Gamma}$ e Γ_f. Caso existam nomes de funções ou variáveis iguais nos dois contextos, na união mantém-se a atribuição de tipo presente em $\bar{\Gamma}$. Por exemplo, suponhamos $\Gamma_{\texttt{f}} = \{\texttt{f} :: \texttt{eArrow}([\texttt{eInt}], \texttt{eInt}), \texttt{g} :: \texttt{eArrow}([\texttt{eFloat}], \texttt{eFloat})\}$ e $\bar{\Gamma} = \{\texttt{f} :: \sigma_1, \texttt{x} :: \texttt{eInt}\}$, então $\bar{\Gamma} \cup \Gamma_{\texttt{f}} = \{\texttt{f} :: \sigma_1, \texttt{x} :: \texttt{eInt}, \texttt{g} :: \texttt{eArrow}([\texttt{eFloat}], \texttt{eFloat})\}$. Consideremos a seguinte definição:

```
1 def f(x):
2     y=x+``a''
3     return y
```

A função `f` tem tipo $\texttt{eArrow}([\texttt{eString}], \texttt{eString})$, pois `x` :: `eString` e `y` :: `eString`.

(Aplicação) Na chamada de uma função o tipo de cada argumento da chamada tem que ser subtipo do tipo do argumento correspondente no tipo da função. Se isto acontece para todos os argumentos, então a chamada tem o tipo do retorno da função.

Consideremos a função anterior, a chamada `f`("`b`") tem tipo `eString`, pois `b` :: `eString` e `eString` <: `eString`. A chamada `f`(`1`) daria erro, uma vez que `1` :: `eInt` e `eInt` não é subtipo de `eString`.

4.2 Sistema de tipos com polimorfismo

Consideremos a função f do Programa 4.1.

```
1 def f(x):
2   return x
```

Programa 4.1

O tipo da função é `eArrow([`σ`],`σ`)`, para qualquer substituição de σ, isto é, tanto podemos substituir σ por um inteiro como por uma palavra, por exemplo. Deste modo, σ representa um conjunto de tipos (esquema de tipos). Um tipo que represente um conjunto de tipos é chamado tipo polimórfico.

No caso da função f o esquema de tipos que representa todos os tipos que podemos ter para esta função é $\forall\sigma.\texttt{eArrow}([\sigma], \sigma)$.

4.2.1 Sintaxe

A sintaxe da linguagem mantém-se a mesma da Secção 4.1.1.

4.2.2 Tipos

Os tipos para este sistema são uma extensão do conjunto definido na secção 4.1.2, com esquemas de tipos.

Os tipos definem-se então pela seguinte gramática:

$$\begin{aligned} \eta ::= \quad & \tau \\ & \mid \ \texttt{eAll}([\sigma_1, ..., \sigma_n], \texttt{eArrow}([\tau_1, ..., \tau_n], \alpha)) \end{aligned}$$

Temos que separar o tipo correspondente ao esquema de tipos dos outros tipos, porque não podemos ter uma função com tipo $\forall\texttt{x.x} \rightarrow \forall\texttt{y.y}$.

O tipo da função `f` do Programa 4.1 representa-se neste sistema de tipos do seguinte modo: $\texttt{eAll}([\sigma], \texttt{eArrow}([\sigma], \sigma))$.

A introdução de esquemas de tipos permite generalizar as funções. A função `f` poderia ser aplicada tanto a inteiros como a qualquer outro tipo de dados. Caso não existissem esquemas de tipos teríamos que definir várias funções que tinham o

mesmo comportamento de `f`, apenas tinham tipos mais específicos: `eArrow([eInt], eInt)`, `eArrow([eString], eString)`, por exemplo.

4.2.3 Subtipos

As regras para os subtipos são as mesma definidas na Secção 4.1.3.

4.2.4 Regras do sistema de tipos

As regras de inferência definidas na Secção 4.1.4 mantém-se neste sistema de tipos. Este conjunto é estendido com duas novas regras:

$$\frac{\bar{\Gamma} = \{\mathtt{e_i} :: \tau_\mathtt{i}\} \qquad 1 \leq \mathtt{i} \leq \mathtt{n} \qquad \bar{\Gamma} \cup \Gamma_\mathtt{f} \vdash \mathtt{e} :: \alpha}{\Gamma'' \vdash \mathtt{def\ f(e_1, ..., e_n) : e\ } :: \ \mathtt{eAll}([\tau_\mathtt{i} \in \mathtt{TVar}], \mathtt{eArrow}([\tau_1, ..., \tau_\mathtt{n}], \alpha))}$$

(Generalização)

$$\Gamma'' = \Gamma \cup \mathtt{f} :: \mathtt{eAll}([\sigma_1, ..., \sigma_\mathtt{n}], \mathtt{eArrow}([\tau_1, ..., \tau_\mathtt{n}], \alpha))$$

$$\frac{\Gamma \vdash \mathtt{f} :: \mathtt{eAll}([\sigma_1, ..., \sigma_\mathtt{n}], \mathtt{eArrow}([\tau_1, ..., \tau_\mathtt{n}], \alpha)) \qquad \Gamma \vdash \bar{\mathtt{e}_\mathtt{i}} :: \alpha_\mathtt{i} \qquad \alpha_\mathtt{i} <: \tau_\mathtt{i} \qquad 1 \leq \mathtt{i} \leq \mathtt{n}}{\Gamma \vdash \mathtt{f}(\bar{\mathtt{e}_1}, ..., \bar{\mathtt{e}_\mathtt{n}}) \ :: \ \alpha}$$

(Aplicação)

Vamos ver cada uma destas regras com mais pormenor:

(Generalização) O tipo da generalização é `eAll(lista,tipo)`. Inferimos o tipo para cada argumento da função assim como para o seu corpo. Destes tipos, os que são variáveis de tipo formam uma lista que é colocada na primeira posição do par (lista, tipo). Na segunda posição deste par é colocado o tipo inferido para a função, tal como na regra (Definição de Função), definida na Secção 4.1.4. Note-se que as funções presentes em Γ_f podem ser polimórficas ou não polimórficas.

Tomemos a função `f` definida no Programa 4.1. Ao inferirmos tipo para `x`, verificamos que é uma variável de tipo (σ). Deste modo, a função `f` recebe a variável de tipo σ e retorna essa mesma variável de tipo. Logo, o tipo de f é `eAll([`σ`], eArrow([`σ`],`σ`))`.

(Aplicação) Semelhante à aplicação definida em 4.1.4.

4.3 Sistema de tipos com polimorfismo e classes

4.3.1 Sintaxe

A sintaxe definida na Secção 4.1.1 é estendida com as seguintes regras:

$$\texttt{e} ::= \ \ldots \ \mid \ \texttt{class c}() : \ [\texttt{e}_1, ..., \texttt{e}_\texttt{n}] \mid \texttt{c} \ (\texttt{e}_1, ..., \texttt{e}_\texttt{n}) \mid \texttt{e.m} \ (\texttt{e}_1, ..., \texttt{e}_\texttt{n}) \mid \texttt{\_\_init\_\_}(\texttt{e}_1, ..., \texttt{e}_\texttt{n})$$

onde,

$\texttt{c} \in$ nomes de classes

$\texttt{m} \in$ nomes de métodos

4.3.2 Tipos

Consideremos o contexto local a uma classe, Ω, definido do seguinte modo:

$$\Omega \ ::= \ \{\texttt{m}_0 :: \eta_0, ..., \texttt{m}_\texttt{n} :: \eta_\texttt{n}\}$$

O conjunto de tipos para este sistema é uma extensão ao conjunto definido na Secção 4.2.2:

$$\tau, \alpha ::= ... \mid \texttt{eClass}(\texttt{c}, \Omega) \mid \texttt{eCCla}(\texttt{x}, [\texttt{c}_1, ..., \texttt{c}_\texttt{n}]) \mid \texttt{eCv}(\texttt{x}, [\tau_1, ..., \tau_\texttt{n}])$$

O tipo `eClass` representa uma classe e o seu contexto. Este contexto é onde os seus métodos e atributos estão definidos. Consideremos a seguinte definição da classe `nova`:

```
1 class nova():
2   def __init__(self,a):
3     self.b=a
4   def po(self):
5     b=self.b
6     return b
```

Esta classe tem tipo $\texttt{eClass}(\texttt{nova}, \{\texttt{self.b} :: \sigma, \texttt{po} :: \texttt{eAll}([\sigma], \texttt{eArrow}([], \sigma)), \texttt{\_\_init\_\_} :: \texttt{eAll}([\sigma], \texttt{eArrow}([\sigma], \texttt{eNone}))\})$. Como já foi referido na Secção 3.1.1, o `self` representa a própria classe, e por essa razão não consideramos o `self` como argumento, na definição dos métodos. Por isso, é que o tipo dos argumentos do método `po` é uma lista vazia, ou seja, não existem argumentos.

O método `__init__` não pode retornar nada ou apenas pode retornar a própria classe. No entanto, sempre que é chamado cria uma instância da classe.

O tipo `eCCla` representa um conjunto de classes enquanto que `eCv` representa um conjunto de valores. Consideremos o Programa 4.2:

```
1  class nova():
2    def __init__(self,a):
3      self.b=a
4    def po(self):
5        b=self.b
6        return b
7
8  class velha():
9      def __init__(self,s):
10       self.d=s
11     def po(self):
12       x=self.d
13     def ra(self,x):
```

```
14        return x
15
16 def f (x0):
17   y=x0(1).po()
18   return y
```

Programa 4.2

O tipo da classe `nova` já conhecemos, pois é o mesmo do exemplo anterior. O tipo da classe `velha` é $\texttt{eClass}(\texttt{velha}, \{\texttt{self.d} = \sigma_1,\ \texttt{po} :: \texttt{eArrow}([], \texttt{eNone}), \texttt{ra} :: \texttt{eAll}([\sigma_2], \texttt{eArrow}([\sigma_2], \sigma_2)),\ \texttt{\_\_init\_\_} :: \texttt{eAll}([\sigma_1], \texttt{eArrow}([\sigma_1], \texttt{eNone}))\})$. O tipo da função `f` é mais difícil de concluir, uma vez que apesar de sabermos que `x0` é o nome de uma classe que possui o método `po` de aridade 0, e em que o `__init__` recebe um único argumento, temos duas classes que respeitam estas condições. Assim, dizemos que `x0` tem tipo $\texttt{eCCla}(\texttt{x0}, [\texttt{velha}, \texttt{nova}])$. O tipo de retorno da função `f`, é o tipo de retorno do método `po`. Como não sabemos qual a classe, não sabemos qual o método `po` que está a ser chamado. Assim, o tipo de retorno da função é $\texttt{eCv}(\texttt{x0}, [\texttt{eNone}, \texttt{eInt}])$. Quando a lista do conjunto de valores tem apenas um tipo, por exemplo $\texttt{eCv}(\texttt{x}, [\texttt{eNone}])$, o que é retornado é o tipo que está na lista, no exemplo, `eNone`.

No Programa 4.2, todas as classes estão definidas antes da definição da função. No entanto, podíamos ter um programa em que isso não acontecesse. Consideremos o seguinte exemplo:

```
1  class nova():
2    def __init__(self,a):
3      self.b=a
4    def po(self):
5        b=self.b
6        return b
7
8  def f (x0):
9    y=x0(1).po()
10   return y
11
```

```
12 class velha():
13     def __init__(self,s):
14       self.d=s
15     def po(self):
16       x=self.d
17     def ra(self,x):
18       return x
```

Neste caso, concluirmos o tipo para `x0` no momento da definição de `f` seria difícil, pois ainda não conhecíamos a classe `velha`. Assim, no nosso sistema assumimos que todas as classes estão definidas no início do programa.

4.3.3 Subtipos

As regras de subtipificação definidas na Secção 4.1.3, têm que ser estendidas para este sistema. As novas regras são:

$$\texttt{eClass}(\texttt{c},\texttt{a}) <: \texttt{eClass}(\texttt{c},\texttt{b}) \qquad \text{(Class)}$$

$$\frac{\texttt{c} \in \texttt{l}}{\texttt{eClass}(\texttt{c},\texttt{a}) <: \texttt{eCCla}(\texttt{id},\texttt{l})} \qquad \text{(ConjClass)}$$

Uma classe é subtipo de um conjunto de classes se pertence a esse conjunto (ConjClass). Por exemplo, $\texttt{eClass}(\texttt{velha},\ \{\texttt{self.d} = \sigma_1, \texttt{po} :: \texttt{eArrow}([], \texttt{eNone}), \texttt{ra} :: \texttt{eAll}([\sigma_2], \texttt{eArrow}([\sigma_2], \sigma_2)),\ \texttt{\_\_init\_\_} ::\ \texttt{eAll}([\sigma_1], \texttt{eArrow}([\sigma_1], \texttt{eNone}))\}) <:\ \texttt{eCCla}(\texttt{x0}, [\texttt{velha}, \texttt{nova}])$.

4.3.4 Regras do sistema de tipos

As regras de inferência para este sistema de tipos são uma extensão das regras definidas na Secção 4.2.4.

Suponhamos um subconjunto do contexto que apenas contém as atribuições de tipo correspondentes a funções e classes:

$$\Gamma_{\texttt{fc}} ::= \{\texttt{t}_\texttt{i} :: \eta_\texttt{i} \ \in \ \Gamma : \ \eta_\texttt{i} \ \texttt{is eArrow(a, b)} \ \mid \ \eta_\texttt{i} \ \texttt{is eAll(c, d)} \ \mid \ \eta_\texttt{i} \ \texttt{is eClass(e, f)}\}$$

As novas regras de inferência são:

$$\frac{\Gamma_{\texttt{fc}} \subseteq \Gamma \qquad \Gamma_{\texttt{fc}} \vdash \texttt{e}_1 :: \eta_1, \ ..., \ \Gamma_{\texttt{fc}} \vdash \texttt{e}_\texttt{n} :: \eta_\texttt{n}}{\Gamma \vdash \texttt{class c}() : [\texttt{e}_1, ..., \texttt{e}_\texttt{n}] \ :: \ \texttt{eClass(c}, \{\texttt{m}_1 :: \eta_1, ..., \texttt{m}_\texttt{n} :: \eta_\texttt{n}\})} \ (\textsc{DefCla})$$

$$\frac{\Gamma \vdash \texttt{c} :: \texttt{eClass(c}, \Omega) \qquad \Gamma, \ \Omega \vdash \texttt{\_\_init\_\_(e}_1, ..., \texttt{e}_\texttt{n}) :: \texttt{eNone}()}{\Gamma \vdash \texttt{c(e}_1, ..., \texttt{e}_\texttt{n}) :: \texttt{eClass(c}, \Omega)} \ (\textsc{Inst1})$$

$$\frac{\Gamma \vdash \texttt{c} :: \texttt{eClass(c}, \Omega) \qquad \Gamma, \ \Omega \vdash \texttt{\_\_init\_\_(e}_1, ..., \texttt{e}_\texttt{n}) :: \texttt{eClass(c}, \Omega)}{\Gamma \vdash \texttt{c(e}_1, ..., \texttt{e}_\texttt{n}) :: \texttt{eClass(c}, \Omega)} \ (\textsc{Inst2})$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{c(e}_1, ..., \texttt{e}_\texttt{n}) :: \texttt{eClass(c}, \Omega) \qquad \Omega \vdash \texttt{m}(\tau_1, ..., \tau_\texttt{n}) :: \eta \\ \Gamma \vdash \bar{\texttt{e}_\texttt{i}} :: \alpha_\texttt{i} \qquad \alpha_\texttt{i} <: \tau_\texttt{i} \qquad 1 \leq \texttt{i} \leq \texttt{n}\end{array}}{\Gamma \vdash \texttt{c(e}_1, ..., \texttt{e}_\texttt{n})\texttt{.m}(\bar{\texttt{e}_1}, ..., \bar{\texttt{e}_\texttt{n}}) :: \eta} \ (\textsc{Acm1})$$

$$\frac{\Gamma \vdash \texttt{c(e}_1, ..., \texttt{e}_\texttt{n}) :: \texttt{eClass(c}, \Omega) \qquad \Omega \vdash \texttt{m} :: \eta}{\Gamma \vdash \texttt{c(e}_1, ..., \texttt{e}_\texttt{n})\texttt{.m} :: \eta} \ (\textsc{Acm2})$$

(DefCla) Quando estamos perante a definição de uma classe, inferimos o tipo ($\eta_\texttt{i}$) de cada um seus métodos ($\texttt{m}_\texttt{i}$). Estes tipos são guardados num contexto local à classe, que fará parte do seu tipo. Como já vimos, no Programa 4.2 a função `nova` tem tipo `eClass( nova,` $\{$`self.b` $:: \sigma$, `po :: eAll(`$[\sigma]$`, eArrow(`$[], \sigma$`)),` `__init__ :: eAll(`$[\sigma]$`, eArrow(`$[\sigma]$`, eNone))`$\}$`)`.

(Inst*) Quando criamos uma instância de uma classe, é invocada a função `__init__` presente no contexto local à classe. O `__init__` só pode retornar a própria classe ou

eNone. Em qualquer dos casos, o tipo da instância é sempre o tipo da classe. Por exemplo, continuando a usar o Programa 4.2, nova(1) :: eClass(nova, {self.b :: eInt, po :: eArrow([], eInt), __init__ :: eArrow([eInt], eNone)}).

(ACM*) Os métodos das classes podem ter ou não argumentos. Quando o método invocado tem argumentos, invocar o método é o mesmo que chamar uma função, tendo em conta que o contexto é o local à instância da chamada. Por exemplo, nova(1).po() :: eInt pois, como vimos no item anterior, no contexto local da instância nova(1), o método po tem tipo eArrow([], eInt). Quando o método não tem argumentos, apenas precisamos de saber qual o seu tipo no contexto local da instância e retorná-lo. Por exemplo, nova(1).b :: eInt.

No corpo de uma função, podemos ter uma instanciação de uma classe que ainda não conhecemos, tal como acontece na função f do Programa 4.2. As regras de inferência seguintes são aplicadas nessas situações.

Suponhamos um dicionário que para cada método indica quais as classes que têm esse método. Vamos definir esse dicionário do seguinte modo:

$$\texttt{overl} = \{\texttt{m}_1 : [\texttt{c}_1, ..., \texttt{c}_t], ..., \texttt{m}_n : [\bar{\texttt{c}_1}, ..., \bar{\texttt{c}_r}]\}$$

Por exemplo, após a definição das classes no Programa 4.2, temos que overl = {__init__ : [velha, nova], b : [nova], d : [velha], po : [velha, nova], ra : [velha]}.

$$\frac{\begin{array}{c} \texttt{l} = \texttt{overl}[\texttt{m}] \\ \Gamma \vdash \texttt{c}_1(\bar{\texttt{e}_1}, ..., \bar{\texttt{e}_j}).\texttt{m}(\texttt{e}_1, ..., \texttt{e}_i) :: \tau_1, \ \ ..., \ \ \Gamma \vdash \texttt{c}_t(\bar{\texttt{e}_1}, ..., \bar{\texttt{e}_j}).\texttt{m}(\texttt{e}_1, ..., \texttt{e}_i) :: \tau_t \\ \Gamma \vdash \texttt{x} :: \sigma \qquad \texttt{l}' = [\tau_1, ..., \tau_t] \end{array}}{\Gamma \vdash \texttt{x}(\bar{\texttt{e}_1}, ..., \bar{\texttt{e}_j}).\texttt{m}(\texttt{e}_1, ..., \texttt{e}_i) \ :: \ \texttt{eCv}(\texttt{x}, \texttt{l}')} \quad (\text{AM11})$$

$$\frac{\begin{array}{c}\mathtt{l} = \mathtt{overl}[\mathtt{m}] \\ \Gamma \vdash \mathtt{c_1}(\bar{\mathtt{e_1}}, ..., \bar{\mathtt{e_j}}).\mathtt{m}(\mathtt{e_1}, ..., \mathtt{e_i}) :: \tau_1,\ ...,\ \Gamma \vdash \mathtt{c_t}(\bar{\mathtt{e_1}}, ..., \bar{\mathtt{e_j}}).\mathtt{m}(\mathtt{e_1}, ..., \mathtt{e_i}) :: \tau_\mathtt{t} \\ \Gamma \vdash \mathtt{x} :: \mathtt{eClass}(\mathtt{c}, \Omega) \qquad [\tilde{\mathtt{c}}_1, ..., \tilde{\mathtt{c}}_\mathtt{n}] = [\mathtt{c_1}, ..., \mathtt{c_t}] \cap \mathtt{c} \qquad \mathtt{l}' = [\tau_1, ..., \tau_\mathtt{n}]\end{array}}{\Gamma \vdash \mathtt{x}(\bar{\mathtt{e_1}}, ..., \bar{\mathtt{e_j}}).\mathtt{m}(\mathtt{e_1}, ..., \mathtt{e_i}) \ :: \ \mathtt{eCv}(\mathtt{x}, \mathtt{l}')} \quad (\textsc{Am}12)$$

$$\frac{\begin{array}{c}\mathtt{l} = \mathtt{overl}[\mathtt{m}] \\ \Gamma \vdash \mathtt{c_1}(\bar{\mathtt{e_1}}, ..., \bar{\mathtt{e_j}}).\mathtt{m}(\mathtt{e_1}, ..., \mathtt{e_i}) :: \tau_1,\ ...,\ \Gamma \vdash \mathtt{c_t}(\bar{\mathtt{e_1}}, ..., \bar{\mathtt{e_j}}).\mathtt{m}(\mathtt{e_1}, ..., \mathtt{e_i}) :: \tau_\mathtt{t} \\ \Gamma \vdash \mathtt{x} :: \mathtt{eCCla}(\mathtt{id}, \mathtt{li}) \qquad [\tilde{\mathtt{c}}_1, ..., \tilde{\mathtt{c}}_\mathtt{n}] = [\mathtt{c_1}, ..., \mathtt{c_t}] \cap \mathtt{li} \qquad \mathtt{l}' = [\tau_1, ..., \tau_\mathtt{n}]\end{array}}{\Gamma \vdash \mathtt{x}(\bar{\mathtt{e_1}}, ..., \bar{\mathtt{e_j}}).\mathtt{m}(\mathtt{e_1}, ..., \mathtt{e_i}) \ :: \ \mathtt{eCv}(\mathtt{x}, \mathtt{l}')} \quad (\textsc{Am}13)$$

(Am1*) Quando queremos invocar um método que ainda não sabemos a que instância (`x`) pertence, invocámo-lo em todas as instâncias onde ele existe, e guardamos os tipos retornados numa lista. Para saber quais são estas instâncias recorremos ao dicionário `overl`. O tipo de retorno da invocação é sempre um conjunto de valores, no entanto o conjunto depende do que já sabemos sobre `x`: se ele é uma variável de tipo então o conjunto de valores é a lista dos tipos inferidos; se é uma classe `c` então o conjunto é a intersecção da lista dos tipos inferidos com o tipo do método, no contexto da classe `c`; se é um conjunto de classes, interseccionámos as classes do conjunto com as invocadas e retornamos apenas o tipo resultante da invocação das classes pertencentes à intersecção.

Tomemos novamente como exemplo o Programa 4.2. A primeira regra é aplicada na função `f`, pois `x0` :: σ. Recorrendo ao `overl` sabemos que x0 tanto pode representar a classe `velha` como a classe `nova`. Invocamos o método em ambas, e ficamos a saber o seu tipo: `nova(1).po()` :: `eInt` e `velha(1).po()` :: `eNone`. Logo, o valor de retorno da invocação é `eCv(x0, [eNone, eInt])`.

Adicionemos agora ao Programa 4.2 as duas funções seguintes:

```
1 def g(x):
2   t=x(1).ra(1)
3   return x(1).po()
4
5 def h(x):
6   t=x(1).po()
7   return x(1).ra(1)
```

Na função g, ao invocarmos o método ra, concluímos que x(1).ra(1) :: eCv(x, [eInt]), ou seja, x(1).ra(1) :: eInt. Como consequência da aplicação desta regra, x fica com o tipo eClass(velha, {self.d = Var(s), po :: eArrow([], eNone), ra :: eArrow ([eInt], eInt)), __init__ :: eArrow([eInt], eNone)}). Quando, de seguida, invocamos o método po a regra a aplicar é a (AM12), uma vez que x é uma classe. Ao consultarmos o dicionário overl verificamos que overl[po] = [velha, nova], logo inferimos o tipo do método em ambas as classes: velha(1).po() :: eNone e nova(1).po() :: eInt. Como a classe velha, que é a classe que o x representa, é uma das classes que tem o método po, o tipo que interessa é o correspondente a velha(1).po(). Assim, x(1).po() :: eCv(x, [eNone]), ou seja, x(1).po() :: eNone.

Na função h, x(1).po() :: eCv(x, [eNone, eInt]), pela aplicação da regra (AM11). Esta aplicação faz também com que x fique com tipo eCCla(x, [velha, nova]). Ao invocarmos, de seguida, o método ra, é aplicada a regra (AM13), uma vez que x é um conjunto de classes. Ao consultarmos overl, verificamos que overl[ra] = velha, e desta forma $[\tilde{c}_1]$ = [velha]. Logo $l' = [eInt]$, e por consequência x(1).ra(1) :: eCv(x, [eInt]), ou seja, x(1).ra(1) :: eInt.

$$\frac{\Gamma \vdash c_1.m(e_1, ..., e_i) :: \tau_1, \ ..., \ \Gamma \vdash c_t.m(e_1, ..., e_i) :: \tau_t \qquad l = overl[m] \qquad \Gamma \vdash x :: \sigma \qquad l' = [\tau_1, ..., \tau_t]}{\Gamma \vdash x.m(e_1, ..., e_i) \ :: \ eCv(x, l')} \quad \text{(AM21)}$$

$$\frac{\begin{array}{c}\mathtt{l} = \mathtt{overl}[\mathtt{m}] \\ \Gamma \vdash \mathtt{c_1.m}(\mathtt{e_1}, ..., \mathtt{e_i}) :: \tau_1,\ ...,\ \Gamma \vdash \mathtt{c_t.m}(\mathtt{e_1}, ..., \mathtt{e_i}) :: \tau_t \\ \mathtt{andalso}\Gamma \vdash \mathtt{x} :: \mathtt{eClass}(\mathtt{c}, \Omega)\ \ [\tilde{\mathtt{c}}_1, ..., \tilde{\mathtt{c}}_n] = [\mathtt{c_1}, ..., \mathtt{c_t}] \cap \mathtt{c} \qquad \mathtt{l}' = [\tau_1, ..., \tau_n]\end{array}}{\Gamma \vdash \mathtt{x.m}(\mathtt{e_1}, ..., \mathtt{e_i})\ ::\ \mathtt{eCv}(\mathtt{x}, \mathtt{l}')} \quad (\text{AM22})$$

$$\frac{\begin{array}{c}\mathtt{l} = \mathtt{overl}[\mathtt{m}] \\ \Gamma \vdash \mathtt{c_1.m}(\mathtt{e_1}, ..., \mathtt{e_i}) :: \tau_1,\ ...,\ \Gamma \vdash \mathtt{c_t.m}(\mathtt{e_1}, ..., \mathtt{e_i}) :: \tau_t \\ \Gamma \vdash \mathtt{x} :: \mathtt{eCCla}(\mathtt{id}, \mathtt{li}) \qquad [\tilde{\mathtt{c}}_1, ..., \tilde{\mathtt{c}}_n] = [\mathtt{c_1}, ..., \mathtt{c_t}] \cap \mathtt{li} \qquad \mathtt{l}' = [\tau_1, ..., \tau_n]\end{array}}{\Gamma \vdash \mathtt{x.m}(\mathtt{e_1}, ..., \mathtt{e_i})\ ::\ \mathtt{eCv}(\mathtt{x}, \mathtt{l}')} \quad (\text{AM23})$$

(AM2*) Semelhante às regras anteriores, no entanto a instância não possui argumentos. Estes são os mesmos que aquando da definição. Suponhamos que a seguinte função é adicionada ao Programa 4.2:

```
1 def t(x):
2   return x.po()
```

Recorrendo ao dicionário `overl` sabemos que as classes que têm o método `po` são a classe `nova` e `velha`. Como não temos argumentos na instância, `x` :: $\mathtt{eCCla}(\mathtt{x}, [\mathtt{velha}, \mathtt{nova}])$ e $\mathtt{x.po}() :: \mathtt{eCv}(\mathtt{x}, [\mathtt{eNone}, \sigma])$, pois o método `po` retorna `eNone` quando a instância é a classe `velha`, e σ quando é a classe `nova`.

$$\frac{\begin{array}{c}\mathtt{l} = \mathtt{overl}[\mathtt{m}] \\ \Gamma \vdash \mathtt{c_1}(\bar{\mathtt{e}_1}, ..., \bar{\mathtt{e}_j}) :: \mathtt{eClass}(\mathtt{c_1}, \Omega),\ ..., \Gamma \vdash \mathtt{c_t}(\bar{\mathtt{e}_1}, ..., \bar{\mathtt{e}_j}) :: \mathtt{eClass}(\mathtt{c_t}, \Omega') \\ \Omega \vdash \mathtt{m} :: \tau_1, ..., \Omega' \vdash \mathtt{m} :: \tau_t \qquad \Gamma \vdash \mathtt{x} :: \sigma \qquad \mathtt{l}' = [\tau_1, ..., \tau_t]\end{array}}{\Gamma \vdash \mathtt{x}(\bar{\mathtt{e}_1}, ..., \bar{\mathtt{e}_j}).\mathtt{m}\ ::\ \mathtt{eCv}(\mathtt{x}, \mathtt{l}')} \quad (\text{AM31})$$

$$\frac{\begin{array}{c}\mathtt{l} = \mathtt{overl}[\mathtt{m}] \\ \Gamma \vdash \mathtt{c}_1(\bar{\mathtt{e}_1}, ..., \bar{\mathtt{e}_j}) :: \mathtt{eClass}(\mathtt{c}_1, \Omega),\ ..., \Gamma \vdash \mathtt{c}_t(\bar{\mathtt{e}_1}, ..., \bar{\mathtt{e}_j}) :: \mathtt{eClass}(\mathtt{c}_t, \bar{\Omega}) \\ \Omega \vdash \mathtt{m} :: \tau_1,\ ...,\ \bar{\Omega} \vdash \mathtt{m} :: \tau_t \\ \Gamma \vdash \mathtt{x} :: \mathtt{eClass}(\mathtt{c}, \Omega') \qquad [\tilde{\mathtt{c}}_1, ..., \tilde{\mathtt{c}}_n] = [\mathtt{c}_1, ..., \mathtt{c}_t] \cap \mathtt{c} \qquad \mathtt{l}' = [\tau_1, ..., \tau_n\end{array}}{\Gamma \vdash \mathtt{x}(\bar{\mathtt{e}_1}, ..., \bar{\mathtt{e}_j}).\mathtt{m} \ ::\ \mathtt{eCv}(\mathtt{x}, \mathtt{l}')} \quad (\text{AM32})$$

$$\frac{\begin{array}{c}\mathtt{l} = \mathtt{overl}[\mathtt{m}] \\ \Gamma \vdash \mathtt{c}_1(\bar{\mathtt{e}_1}, ..., \bar{\mathtt{e}_j}) :: \mathtt{eClass}(\mathtt{c}_1, \Omega), ... \ , \Gamma \vdash \mathtt{c}_t(\bar{\mathtt{e}_1}, ..., \bar{\mathtt{e}_j}) :: \mathtt{eClass}(\mathtt{c}_t, \Omega') \\ \Omega \vdash \mathtt{m} :: \tau_1,\ ...,\ \Omega \vdash \mathtt{m} :: \tau_t \\ \Gamma \vdash \mathtt{x} :: \mathtt{eCCla}(\mathtt{x}, \mathtt{li}) \qquad [\tilde{\mathtt{c}}_1, ..., \tilde{\mathtt{c}}_n] = [\mathtt{c}_1, ..., \mathtt{c}_t] \cap \mathtt{li} \qquad \mathtt{l}' = [\tau_1, ..., \tau_n\end{array}}{\Gamma \vdash \mathtt{x}(\bar{\mathtt{e}_1}, ..., \bar{\mathtt{e}_j}).\mathtt{m} \ ::\ \mathtt{eCv}(\mathtt{x}, \mathtt{l}')} \quad (\text{AM33})$$

(AM3*) Semelhante às regras (AM1*), mas com métodos sem argumentos. Consideremos a adição da seguinte função ao Programa 4.2:

```
1 def g(x):
2   return x(1).b
```

A variável `x` só pode ser a classe nova, uma vez que só esta tem o método `b`. Assim, sabemos que $\mathtt{x}(1) :: \mathtt{eClass}(\ \mathtt{nova},\ \{\mathtt{self.b} :: \mathtt{eInt},\ \mathtt{po} ::\ \mathtt{eArrow}([], \mathtt{eInt}),\ \_\_\mathtt{init}\_\_ ::\ \mathtt{eArrow}([\mathtt{eInt}], \mathtt{eNone})\})$. Logo, $\mathtt{x}(1).\mathtt{b} :: \mathtt{eInt}$.

$$\frac{\begin{array}{c}\mathtt{l} = \mathtt{overl}[\mathtt{m}] \\ \Gamma \vdash \mathtt{c}_1 :: \mathtt{eClass}(\mathtt{c}_1, \Omega),\ ..., \Gamma \vdash \mathtt{c}_t :: \mathtt{eClass}(\mathtt{c}_t, \Omega') \\ \Omega \vdash \mathtt{m} :: \tau_1,\ ..., \Omega' \vdash \mathtt{m} :: \tau_t \qquad \Gamma \vdash \mathtt{x} :: \sigma \qquad \mathtt{l}' = [\tau_1, ..., \tau_t]\end{array}}{\Gamma \vdash \mathtt{x.m} \ ::\ \mathtt{eCv}(\mathtt{x}, \mathtt{l}')} \quad (\text{AM41})$$

$$\frac{\begin{array}{c}\mathtt{l} = \mathtt{overl}[\mathtt{m}] \\ \Gamma \vdash \mathtt{c}_1 :: \mathtt{eClass}(\mathtt{c}_1, \Omega),\ ...,\ \mathtt{c}_t :: \mathtt{eClass}(\mathtt{c}_t, \bar{\Omega}) \\ \Omega \vdash \mathtt{m} :: \tau_1, ..., \bar{\Omega} \vdash \mathtt{m} :: \tau_t \qquad \Gamma \vdash \mathtt{x} :: \mathtt{eClass}(\mathtt{c}, \Omega') \\ [\tilde{\mathtt{c}}_1, ..., \tilde{\mathtt{c}}_n] = [\mathtt{c}_1, ..., \mathtt{c}_t] \cap \mathtt{c} \qquad \mathtt{l}' = [\tau_1, ..., \tau_n]\end{array}}{\Gamma \vdash \mathtt{x.m} \ ::\ \mathtt{eCv}(\mathtt{x}, \mathtt{l}')} \quad (\textsc{Am}42)$$

$$\frac{\begin{array}{c}\mathtt{l} = \mathtt{overl}[\mathtt{m}] \\ \Gamma \vdash \mathtt{c}_1 :: \mathtt{eClass}(\mathtt{c}_1, \Omega),\ ...,\ \Gamma \vdash \mathtt{c}_t :: \mathtt{eClass}(\mathtt{c}_t, \Omega') \\ \Omega \vdash \mathtt{m} :: \tau_1, ..., \Omega' \vdash \mathtt{m} :: \tau_t \qquad \Gamma \vdash \mathtt{x} :: \mathtt{eCCla}(\mathtt{x}, \mathtt{li}) \\ [\tilde{\mathtt{c}}_1, ..., \tilde{\mathtt{c}}_n] = [\mathtt{c}_1, ..., \mathtt{c}_t] \cap \mathtt{li} \qquad \mathtt{l}' = [\tau_1, ..., \tau_n]\end{array}}{\Gamma \vdash \mathtt{x.m} \ ::\ \mathtt{eCv}(\mathtt{x}, \mathtt{l}')} \quad (\textsc{Am}43)$$

(Am4*) Semelhante às regras Am2*, mas com métodos que não têm argumentos. Consideremos o Programa 4.2 com a seguinte função:

```
1 def r(x):
2   return x.d
```

A variável x só pode representar a classe velha, uma vez que overl[d] = velha. Como a instância não possui argumentos x :: eClass(velha, {self.d = σ_1, po :: eArrow([], eNone), ra :: eAll([σ_2], eArrow([σ_2], σ_2)), __init__ :: eAll([σ_1], eArrow([σ_1], eNone))}). Assim, x.d :: σ_1.

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{f} :: \mathtt{eArrow}([\mathtt{e}_1, ..., \mathtt{e}_n], \alpha) \ \ \Gamma \vdash \mathtt{e}_1 \mathtt{op}\ \bar{\mathtt{e_1}}, ..., \Gamma \vdash \mathtt{e}_n \mathtt{op}\ \bar{\mathtt{e_n}} \\ \Gamma \vdash \ \alpha :: \mathtt{eCv}(\mathtt{id}, \mathtt{lista}) \qquad \Gamma \vdash \mathtt{e}_i :: \mathtt{eCCla}(\mathtt{id}, \mathtt{l}) \\ \Gamma \vdash \bar{\mathtt{e_i}} :: \mathtt{eClass}(\mathtt{c}, \Omega) \qquad \mathtt{l}[\mathtt{j}] = \mathtt{c}\end{array}}{\Gamma \vdash \mathtt{f}(\bar{\mathtt{e_1}}, ..., \bar{\mathtt{e_n}}) \ ::\ \mathtt{lista}[\mathtt{j}]} \quad (\textsc{Apl}1)$$

$$\frac{\Gamma \vdash \mathtt{f} :: \mathtt{eAll}([\mathtt{var}], \mathtt{eArrow}([\mathtt{e_1}, ..., \mathtt{e_n}], \alpha)) \quad \Gamma \vdash \mathtt{e_1 op}\ \bar{\mathtt{e_1}}, ..., \Gamma \vdash \mathtt{e_n op}\ \bar{\mathtt{e_n}} \quad \Gamma \vdash\ \alpha :: \mathtt{eCv}(\mathtt{id}, \mathtt{lista}) \quad \Gamma \vdash \mathtt{e_i} :: \mathtt{eCCla}(\mathtt{id}, \mathtt{l}) \quad \Gamma \vdash \bar{\mathtt{e_i}} :: \mathtt{eClass}(\mathtt{c}, \Omega) \quad \mathtt{l}[\mathtt{j}] = \mathtt{c}}{\Gamma \vdash \mathtt{f}(\bar{\mathtt{e_1}}, ..., \bar{\mathtt{e_n}})\ ::\ \mathtt{lista}[\mathtt{j}]} \text{(APL2)}$$

(APL*) Suponhamos uma função $\mathtt{k} :: \mathtt{eArrow}([\mathtt{eCCla}(\mathtt{x}, [\mathtt{velha}, \mathtt{nova}])], \mathtt{eCv}(\mathtt{x}, [\mathtt{eNone}, \mathtt{eInt}]))$, definida no Programa 4.2. Ao invocarmos $\mathtt{k}(\mathtt{velha})$, temos que verificar se os argumentos da chamada e da definição da função são subtipo um do outro, neste caso $\mathtt{velha} <: \mathtt{eCCla}(\mathtt{x}, [\mathtt{velha}, \mathtt{nova}])$. Como o tipo de retorno da função é um conjunto de valores, isso significa que um dos argumentos é um conjunto de classes, neste caso o único argumento. Como a posição da classe `velha`, argumento da chamada, no conjunto de classes é 0, o tipo retornado vai ser o índice 0 do conjunto de valores, neste caso `eNone`.

Capítulo 5

Implementação da inferência de tipos

Neste capítulo vamos descrever um algoritmo de inferência de tipos, tendo por base o sistema de tipos definido no Capítulo 4. Este algoritmo de inferência de tipos foi implementado na linguagem Python.

5.1 Análise sintáctica

A análise sintáctica do programa para o qual pretendemos fazer a inferência de tipos é efectuado pelo módulo AST do Python. Este módulo define uma sintaxe abstracta para o Python. Esta sintaxe pode ser representada por uma árvore sintáctica abstracta, onde cada nó representa um constructor sintáctico e a raiz o objecto *Module*. A sintaxe é abstracta no sentido em que não representa todos os detalhes que aparecem na sintaxe real. Por exemplo, a instrução `if-then-else` pode ser representada por um único nó. Desta forma, a sintaxe abstracta oferece uma interface de alto nível para o código Python já analisado.

A gramática abstracta do Python pode ser consultada em [Py]. No Apêndice A pode

ser consultado o subconjunto desta gramática, que contém as instruções aceites pelo algoritmo de inferência que desenvolvemos.

Tomemos como exemplo o seguinte fragmento da sintaxe abstracta da linguagem Python:

`stmt` = `FunctionDef`(`identifier name`, `arguments args`, `stmt` ∗ `body`)

|`Return`(`expr`? `value`)

Este exemplo descreve dois tipos diferentes de instruções: a definição de funções e a instrução `Return`. A definição de funções tem três argumentos: o seu nome, uma lista com os argumentos e zero ou mais instruções que compõem o seu corpo. A instrução `Return` tem uma expressão opcional que é o valor a ser retornado.

A árvore sintáctica é o ponto de partida do algoritmo de inferência de tipos.

5.2 Algoritmo de inferência de tipos

O processo de inferência de tipos é implementado numa classe Python designada `Inferenc`. Nesta classe temos:

- um contexto global, designado `amb`, onde são guardadas as atribuições de tipo;
- um dicionário, designado `dic`, que contém a informação acerca das variáveis de tipo que estão a ser utilizadas;
- o dicionário `overl`, que para cada método indica quais as classes que têm esse mesmo método.

A função `subtype`, definida na classe `Inference`, dados dois tipos como argumento, verifica se o primeiro é subtipo do segundo. Os tipos, utilizados no processo de inferência, são implementados como classes Python. A função `infer`, função principal da classe `Inferenc`, procede à inferência de tipos. Para tal, implementa uma

sequência de instruções condicionais, em que cada uma delas representa um construtor da linguagem.

De seguida, apresentamos e explicamos, de forma breve, a implementação do processo de inferência para alguns dos construtores da linguagem.

Número A regra NUM* (página 44), pode ser implementada do seguinte modo:

```
1 if isinstance(node,ast.Num):
2   if type(node.n)== int:
3     return eInt()
4   elif type(node.n)==float:
5     return eFloat()
6   elif type(node.n)==long:
7     return eLong()
```

O teste na linha 1 verifica se o nó que estamos a avaliar é um número. Ao longo de todo a implementação fazemos este teste de forma a saber qual a regra a aplicar. Para sabermos o tipo do número usamos a função `type` do Python, que permite saber qual o tipo de um objecto.

Lista Para a regra LST (página 45) temos a seguinte implementação:

```
1  elif isinstance(node,ast.List):
2    if len(node.elts)==0:
3      return eList(eNone())
4    else:
5      tipoc=self.infer(node.elts[0])
6      for i in range(1,len(node.elts)):
7        tipo= self.infer(node.elts[i])
8        if self.subtype(tipoc,tipo):
9          tipoc = tipo
10       elif self.subtype(tipo,tipoc):
11         pass
```

```
12        else:
13          raise ERRO()
14      return eList(tipoc)
```

Uma lista [1,2.0,3] é representada na árvore sintáctica pelo nó ast.List([1,2.0,3], “”). A lista [1,2.0,3] é o parâmetro `elts` do objecto `ast.List`. Quando este parâmetro não contém qualquer elemento retornamos o tipo `eList(eNone)`. Caso existam elementos temos que inferir o tipo para cada um deles, para tal chamamos a função `infer`. Se os tipos inferidos para os elementos são subtipo uns dos outros, o tipo retornado é eList(τ), onde τ é o tipo do qual todos os outros são subtipo; caso contrário é lançado um erro. Por exemplo, $[1, 2.0, 3] :: eList(eFloat)$, uma vez que $1 :: eInt$, $2.0 :: eFloat$, $3 :: eInt$ e $eInt <: eFloat$.

Quando, por alguma razão, não é possível inferir o tipo (como acontece na linha 13 do exemplo anterior) é lançado um erro, de nome ERRO, que é definido do seguinte modo:

```
1 class ERRO():
2   def __str__(self):
3     return repr(''erro de tipos'')
```

Atribuição A regra Atr* (página 46) é implementada do seguinte modo:

```
1 elif isinstance(node,ast.Assign):
2   try:
3     tipov=self.infer(node.targets[0])
4     tipoa=self.infer(node.value)
5   except ERRO:
6     if isinstance(node.targets[0],ast.Attribute):
7       classe=node.targets[0].value
8       atr=node.targets[0].attr
9       if isinstance(classe, ast.Name):
```

```
10          if classe.id==self.sel:
11            node.targets[0] = ast.Name(``self.''+node.targets[0].attr,```
                  )
12      if isinstance(node.targets[0], ast.Name):
13        tipoa=self.infer(node.value)
14        if type(tipoa)==str:
15          tipoa=Var(tipoa)
16        self.amb.update({node.targets[0].id:tipoa})
17        return eNone()
18      else:
19        raise ERRO()
20    if self.subtype(tipov,tipoa):
21      if isinstance(node.targets[0], ast.Name):
22        self.amb.update({node.targets[0].id:tipoa})
23    elif self.subtype(tipoa,tipov):
24      pass
25    else:
26      raise ERRO()
27    return eNone()
```

Uma atribuição é representada na árvore abstracta do Python por `ast.Assign (expr* targets, expr value)`, onde `targets` representa o lado esquerdo da atribuição e `value` o lado direito. Por exemplo a atribuição x=3 é representada por:

```
ast.Assign ([ast.Name (x,```)], ast.Num(3)).
```

Tentamos inferir tipo para `x` e para 3 (linha 3 e 4). Caso a variável x não esteja no contexto, é-lhe atribuído o tipo de 3, ou seja, eInt (linhas 12 a 17).

A atribuição `self.y=2` é representada na árvore por:

```
ast.Assign (ast.Attribute (ast.Name (self,```),
      ast.Name (y,```), ```), ast.Num(2)).
```

Quando estamos perante um atributo em que, tal como no exemplo, a primeira parte é referente à classe corrente tratámos o atributo como uma variável. É o teste na linha 10 que nos permite verificar se estamos perante uma referência à classe corrente. Esta classe é guardada na variável `sel`. No exemplo, `sel` indicaria `self`.

Depois de termos inferido o tipo de ambos os lados da atribuição verificamos se eles são subtipo um do outro (linhas 20 a 24). Caso isso não aconteça é lançado um erro de tipos. Caso nenhum erro seja lançado é retornado o tipo `eNone`.

Operações de comparação Vejamos agora como as operações de comparação definidas pela regra OPC* (página 47), são implementadas:

```
1 elif isinstance(node, ast.Compare):
2   tipo1=self.infer(node.left)
3   tipo2=eNone()
4   for i in range(0,(len(node.comparators))):
5     tipo2=self.infer(node.comparators[i])
6     if self.subtype(tipo1,tipo2)==False:
7       if self.subtype(tipo2,tipo1)==False:
8         raise ERRO()
9     return eBool()
```

As operações de comparação são apresentadas na árvore abstracta por `ast.Compare( expr left, cmpop* ops, expr* comparators)`. Por exemplo, $2 == 3$ representa-se por:

```
ast.Compare (ast.Num(2), [Eq], [ast.Num(3)]).
```

O argumento `comparators` tem que ser uma lista por causa dos casos em que temos mais que uma comparação, por exemplo $2 < 3 < 4$, onde 3 e 4 ficam nesta lista. Verificamos se os tipos dos valores envolvidos na operação (ou operações no último exemplo) são subtipos uns dos outros. Caso isto aconteça retornamos `eBool`; caso contrário é lançado um erro.

Condicional O condicional descrito pela regra COND* (página 50) é implementado da seguinte forma:

```
elif isinstance(node, ast.If):
  tipo1=self.infer(node.test)
  tipo2c=eNone()
  for i in range(0,(len(node.body))):
    tipo2=self.infer(node.body[i])
    if self.subtype(tipo2c,tipo2):
      tipo2c=tipo2
    elif self.subtype(tipo2,tipo2c):
      pass
    else:
      raise ERRO()
  if node.orelse==[]:
    tipo3c=tipo2c
  else:
    tipo3c=eNone()
    for i in range(0,(len(node.orelse))):
      tipo3=self.infer(node.orelse[i])
      if self.subtype(tipo3c,tipo3):
        tipo3c=tipo3
      elif self.subtype(tipo3,tipo3c):
        pass
      else:
        raise ERRO()
  if tipo1.nome==eBool().nome:
    if self.subtype(tipo2c,tipo3c):
      return tipo3c
    elif self.subtype(tipo3c,tipo2c):
      return tipo2c
    else:
      raise ERRO()
  else:
    raise ERRO()
```

O nó da árvore abstracta que representa a instrução condicional é `ast.If (expr test, stmt* body, stmt* orelse)`. Por exemplo, a instrução:

```
if 2==3: return 3 else: return 2
```

é representada por

```
ast.If (ast.Compare (ast.Num(2), [Eq],[ast.Num(3)]),
    [ast.Return(Num(3))], [ast.Return(Num(2))]).
```

Inicialmente inferimos o tipo do campo `test`. De seguida, percorremos a lista do campo `body`, inferindo o tipo de cada elemento. Cada elemento desta lista é uma instrução e a única instrução que retorna um tipo diferente de `eNone` é o `return`. Assim, verificarmos se os tipos das instruções em `body` são subtipo uns dos outros, é garantir que não existem retornos de tipos diferentes dentro do campo `body`.

Caso exista `else` procede-se à inferência de modo semelhante ao descrito acima. Caso contrário, o tipo deste campo é `eNone`.

Depois de inferidos os tipos de todos os campos do objecto `ast.If`, verificamos se o teste tem tipo booleano. Caso isto se verifique os tipos inferidos para os campos `body` e `orelse` têm que ser subtipo um do outro.

Definição de funções A definição de funções é representada na árvore abstracta pelo nó `ast.FunctionDef ( identifier name, arguments args, stmt* body, expr* decorator_ list)`. Consideremos a função seguinte:

```
1 def f(x):
2   return x
```

A função f é representada na árvore abstracta por:

```
ast.FunctionDef(f, ([ast.Name(x,‘‘’’)],[]),
    [ast.Return(ast.Name(x,‘‘’’))], []).
```

Como vimos no capítulo anterior existem duas regras para a definição de funções (DefFunc (página 50) e Generalização (página 53)) dependendo se a função é ou não polimórfica. Em ambas as regras os argumentos são inferidos num novo contexto, ou seja, num contexto inicialmente vazio. Assim, a implementação da inferência dos argumentos da função é:

```
1 n=Inferenc()
2 arg=node.args.args
3 tipoarg=[]
4 for i in range(0, len(arg)):
5   t=n.infer(arg[i])
6   tipoarg.append(t)
```

Para criarmos um novo contexto na implementação, criamos uma nova instância da classe onde efectuamos a inferência (linha 1). Inferimos o tipo para cada argumento, na instância criada (linha 5), e guardamos os tipos inferidos numa lista designada tipoarg (linha 6).

De seguida, inferimos o tipo para o corpo da função. No entanto, este tipo é inferido num contexto resultante da união do contexto onde foram inferidos os argumentos e o contexto que contém as atribuições de tipo correspondentes às funções e às classes. Esta união é implementada do seguinte modo:

```
1 dici = self.amb.copy()
2 func=self.dicfun(dici)
3 for i in n.amb.keys():
4   if func.has_key(i):
5     del func[i]
6 n.amb.update(func)
```

O dicionário `dici` representa o contexto corrente, enquanto que o dicionário `n.amb` representa o contexto onde foi inferido o tipo dos argumentos. A função `dicfun` retorna um dicionário com as atribuições de tipo correspondentes a funções e classes, presentes no dicionário passado como argumento. O ciclo é efectuado para retirar do dicionário `func` as funções e classes presentes no dicionário `n.amb`, uma vez que caso tenhamos o mesmo nome de função ou classe nos dois dicionários, mantém-se na união a atribuição de tipo presente no `n.amb`. Por fim, adicionámos ao dicionário `n.amb` o dicionário `func`.

A inferência do corpo da função é implementada da seguinte forma:

```
1 tipor=eNone()
2 for i in range(0, len(node.body)):
3   tp=n.infer(node.body[i])
4   if n.subtype(tipor,tp):
5     tipor=tp
6   elif n.subtype(tp,tipor):
7     pass
8   else:
9     raise ERRO()
```

A variável `tipor` é inicializada com o tipo `eNone`, e vai ser como que o tipo corrente do corpo da função. Inferimos o tipo de cada instrução presente no corpo da função, e garantimos que o tipo inferido e o `tipor` são subtipo um do outro. A variável `tipor` é alterada para o tipo inferido, caso o seu valor seja subtipo desse tipo inferido.

Após a inferência verificamos se existem variáveis de tipo, para sabermos se estamos perante uma função polimórfica ou não polimórfica. Retornamos o tipo da função dependendo do caso em que estamos:

```
1 if var==[]:
2   self.amb.update({node.name:eArrow(tipoarg,tipor)})
3   return eArrow(tipoarg,tipor)
4 else:
```

```
5   self.amb.update({node.name:eAll(var,eArrow(tipoarg,tipor))})
6   return eAll(var,eArrow(tipoarg,tipor))
```

Definição de classes Podemos criar novos objectos, definindo novas classes. A definição de classes é representada na árvore sintáctica abstracta pelo nó `ast.ClassDef (identifier name, expr* bases, stmt* body, expr *decorator_list)`. Tal como na definição de funções, o tipo dos atributos e métodos da classe é inferido num novo contexto, que apenas contém as funções e classes já definidas no contexto corrente:

```
1 d=self.amb.copy()
2 func=self.dicfun(d)
3 n=Inferenc()
4 n.amb.update(func)
```

O dicionário `overl` indica, para cada método, quais as classes que possuem esse método. Assim, para cada método de uma classes temos que o adicionar ao dicionário `overl`, dizendo a classe onde está inserido:

```
1  nome=node.name
2  for i in range(0, len(node.body)):
3    if isinstance(node.body[i],ast.FunctionDef):
4      if self.overl.has_key(node.body[i].name):
5        l=n.overl[node.body[i].name]
6        li=[(nome,tipo)]+l
7        n.overl.update({node.body[i].name:li})
8      else:
9        n.overl.update({node.body[i].name:[(nome,tipo)]})
10
11   self.overl.update(n.overl)
```

Os métodos não são mais do que definições de funções dentro de definições de classes, por isso fazemos o teste na linha 3. Para adicionarmos o método corrente ao dicionário verificamos se ele já existe (linhas 4 a 9): se sim, apenas adicionamos o nome da classe que está a ser definida à lista de classes já existente em `overl`; caso contrário, adicionámos o nome do método a `overl` e o nome da classe. A variável `tipo` representa o tipo inferido para o método. Este dicionário overl é adicionado ao dicionário `overl` do contexto global.

Por fim é adicionado ao contexto global o tipo da classe e retornado esse mesmo tipo:

```
1 self.amb.update({nome:eClass(nome,n.amb)})
2 return eClass(nome,n.amb)
```

O nó ast.Call(expr func, expr* args, keyword* keywords, expr? starargs, expr? kwargs) da árvore sintáctica pode representar a instanciação de uma classe, o acesso a métodos de uma classe, ou a chamada a uma função.

Instanciação de classe A regra INST* (página 58) é implementada do seguinte modo:

```
1 if tipof.nome == eClass(''','''').nome:
2   ambclass=tipof.amb.copy()
3   fun=ambclass[''__init__'']
4   self.amb.update({''__init__'':fun})
5   t=self.infer(ast.Call(ast.Name(''__init__'',''''), node.args,''',''''
        ,''''))
6   del self.amb[''__init__'']
```

A instanciação de uma classe é equivalente ao acesso ao método `__init__` dessa classe. Este método apenas pode retornar `self` ou não retornar, caso contrário deve ser lançado um erro:

```
if t==self.sel:
  return eClass(tipof.id,ambclass)
elif t.nome==eNone().nome:
  return eClass(tipof.id,ambclass)
else:
  raise ERRO()
```

Acesso a métodos Como vimos no capítulo anterior os acessos a métodos de classes podem ser feitos a classes conhecidas (ACM*), ou a classes que são argumentos de funções, e que por isso são representadas por variáveis (AM*). Se conhecemos a classe, para aceder ao seu método basta copiá-lo do contexto local à classe para o contexto corrente. De seguida, inferimos o seu tipo, quando chamado com os argumentos. Por fim, retornamos o tipo inferido.

```
if self.amb[node.func.value.func.id].nome != Var(''').nome:
  cla=self.infer(node.func.value)
  idfunc=node.func.attr
  tipof=cla.amb[idfunc]
  self.amb.update({''funcao'':tipof})
  t=self.infer(ast.Call(ast.Name(''funcao'',''''), node.args,'''','''
      ,'''''))
  del self.amb[''funcao'']
  return t
```

Caso não conheçamos a classe, verificamos no dicionário `overl` quais as classes que possuem o método. Tentamos instanciar cada uma delas, tendo como argumentos os argumentos iniciais. Sempre que a instanciação é possível adicionamos a classe e o seu tipo à lista `li`.

```
var=self.amb[node.func.value.func.id].var
idfunc=node.func.attr
l=self.overl[idfunc]
```

```
4  li=[]
5  for i in range(0, len(l)):
6    try:
7      (c,t)=l[i]
8      d=self.d.dic.copy()
9      cla=self.infer(ast.Call(ast.Name(c,''''), node.func.value.args, ''''
           ,'''',''''))
10     tipof=cla.amb[idfunc]
11     self.d.dic=d.copy()
12     self.amb.update({''funcao'':tipof})
13     t=self.infer(ast.Call(ast.Name(''funcao'',''''), node.args,'''','''
           ,''''))
14     del self.amb[''funcao'']
15     li.append((c,t))
16   except:
17     pass
18 l=[]+li
```

Caso a variável que representa a classe já tenha algum tipo associado temos que interseccionar este tipo com a lista `li`. A função `intersecao`, que recebe como argumento dois tipos, os quais apenas podem ser classes ou conjuntos de classes, define-se do seguinte modo:

```
1  def intersecao(self,l1,l2):
2    l=[]
3    for i in range(0,len(l1)):
4      n1,t1=l1[i]
5      for i in range(0,len(l2)):
6        n2,t2=l2[i]
7        if n1==n2:
8          l.append((n1,t2))
9    return l
```

Caso a lista l tenha tamanho zero é lançado um erro, pois isso significa que não existe qualquer classe que respeite as características impostas. Caso tenha tamanho um, é retornado o tipo do método na classe presente na lista, e a variável passa a representar essa classe. Caso tenha tamanho superior a um, o tipo retornado é o conjunto dos tipos que o método toma, em cada uma das classes presentes na lista, e a variável passa a representar esse conjunto de classes.

```
1  if self.d.last_ref(var)==''':
2    if len(l)==0:
3      raise ERRO()
4    elif len(l)==1:
5      (c,t)=l[0]
6      self.d.actualiza(var,self.amb[c])
7      if type(t)==str:
8        ti=self.amb[c].amb[idfunc]
9        self.amb.update({``funcao'':ti})
10       t=self.infer(ast.Call(ast.Name(``funcao'',````), node.args,````,``
             '',````))                          return t
11   else:
12     self.d.actualiza(var, eCCla(var,l))
13     return eCv(var,l)
14 else:
15   if self.d.last_ref(var).nome == eCCla(````,````).nome:
16     lista=self.d.last_ref(var).lista
17   elif self.d.last_ref(var).nome == eClass(````,````).nome:
18     lista=[(self.d.last_ref(var).id,self.d.last_ref(var).amb[idfunc])]
19   l=self.intersecao(lista,l)
20   if len(l)==0:
21     raise ERRO()
22   elif len(l)==1:
23     (c,t)=l[0]
24     self.d.actualiza(var,self.amb[c])
25     if type(t)==str:
26       ti=self.amb[c].amb[idfunc]
27       self.amb.update({``funcao'':ti})
```

```
28        t=self.infer(ast.Call(ast.Name(``funcao'',````), node.args,````,``
              '',````))
29      return t
30    else:
31      self.d.actualiza(var,eCCla(var,l))
32      return eCv(var,l)
```

Chamada a função A implementação da chamada a uma função depende se esta é não polimórfica ou polimórfica. No primeiro caso, o que temos que fazer é verificar se o tipo dos argumentos recebidos (`arg`) são subtipo dos argumentos na definição da função (`targ`). Se o tipo de um dos argumentos na definição da função é um conjunto de classes e o tipo de retorno(`tipa`) é um conjunto de valores, então verificamos qual a classe dada como argumento, e retornamos o valor correspondente a essa classe. A implementação pode ser feita do seguinte modo:

```
1  elif tipof.nome == eArrow(````,````).nome:
2    targ=tipof.en
3    tipa=tipof.sa
4    arg=node.args
5    if len(targ)==len(arg):
6      for i in range(0,len(arg)):
7        tipo=self.infer(arg[i])
8        if self.subtype(tipo,targ[i])==False: raise ERRO()
9        if targ[i].nome == eCCla(````,````).nome:
10         if tipa.nome == eCv(````,````).nome:
11           if targ[i].id==tipa.id:
12             for j in range(0,len(tipa.lista)):
13               (c,t)=tipa.lista[j]
14               if c == tipo.id:
15                 tipa=t
16                 break
17   else: raise ERRO()
```

Se a função é polimórfica o procedimento é semelhante.

A chamada a uma função `f` pode ser feita dentro da definição de outra função, na qual `f` é argumento. Nestas situações nós podemos não conhecer o tipo de `f`, apenas sabemos que está a ser feita uma chamada e que por isso ele representa uma função. Assim, inferimos o tipo dos argumentos da chamada, e esses serão o tipo dos argumentos na definição da função f. O tipo de retorno destas funções é uma variável de tipo, reconhecida pelo sistema apenas para estas situações, designada `EVAi`, onde i é o número de variáveis de tipo desta forma, já presentes no contexto. Esta situação pode ser implementada da seguinte forma:

```
1  if self.d.last_ref(tipof.var)==’’:
2    tipoarg=[]
3    for i in range (0,len(node.args)):
4      tipoarg.append(self.infer(node.args[i]))
5    i=0
6    nome=‘‘EVA’’
7    while i!=len(self.d.dic):
8      if self.d.dic.has_key(nome):
9        nome=nome+str(i)
10     else:
11       break
12   self.d.adiciona(nome)
13   if self.subtype(tipof, eArrow(tipoarg,Var(nome))):
14     return Var(nome)
15   else:
16     raise ERRO()
17 else:
18   return self.d.last_ref(tipof.var)
```

A chamada a funções ou classes passadas como argumento pode trazer problemas. Consideremos a seguinte função:

```
1 def f(x):
2   y=x(1)
3   return x(1).po()
```

Quando, na função `f`, fazemos a chamada `x(1)` não sabemos se `x` é uma classe ou uma função. Admitimos que é uma função. No entanto, quando chegamos a instrução na linha 3, verificamos que x é uma classe. Na nossa implementação, para que `x` fosse tratado como uma classe a função tinha que ser definida do seguinte modo:

```
1 def f(x):
2   y=x.__init__(1)
3   return x(1).po()
```

Atributos Os atributos são representados na árvore abstracta do Python pelo nó `ast.Attribute(expr value, identifier attr, expr_context ctx)`. Caso o primeiro campo do nó seja o `self`, adicionamos o atributo ao dicionário `overl`, indicando que pertence à classe que está a ser avaliada no momento. Depois inferimos o tipo do atributo. Os atributos desta forma são sempre tratados como variáveis com o nome `self.*`. A implementação pode ser feita da seguinte forma:

```
1 if classe.id==self.sel:
2   nome=``self.''+atr
3   if self.overl.has_key(atr):
4     l=self.overl[atr]
5     li=[(self.curclas,Var(nome))]+l
6     self.overl.update({atr:li})
7   else:
8     self.overl.update({atr:[(self.curclas,Var(nome))]})
9     return self.infer(ast.Name(nome,``''))
```

Caso o primeiro campo do nó seja uma classe, instanciada sem argumentos, retornamos o tipo do atributo no contexto da classe. Note-se que no contexto da classe o atributo estará guardado com `self` no primeiro campo. Assim, temos:

```
1 if self.amb[classe.id].nome==eClass(‘‘’’,‘‘’’).nome:
2   c=self.amb[classe.id]
3   a=c.amb
4   var=c.id
5   at=‘‘self.’’+atr
6   return a[at]
```

Caso a classe seja instanciada com argumentos, isto é, `classe(1,2).atributo`, temos que instanciar a classe e só depois retornar o tipo do atributo na instanciação da classe.

```
1 if self.amb[classe.func.id].nome==eClass(‘‘’’,‘‘’’).nome:
2   c=self.amb[classe.func.id].id
3   cla=self.infer(ast.Call(ast.Name(c,‘‘’’), classe.args, ‘‘’’,‘‘’’,‘‘’’)
        )
4   return cla.amb[‘‘self.’’+atr]
```

Se o primeiro campo do atributo é uma variável de tipo que ainda não instanciamos, a inferência é implementada de forma semelhante à instanciação de classes implementada na página 79 e na página 81.

5.3 Exemplo prático

Suponhamos a definição de duas classes: uma classe `Node` que implementa o nó de uma lista, e os acessos a este; e uma classe `List` que implementa uma lista, e os métodos inserção de um nó numa lista e verificação se uma lista é vazia. O código Python desta implementação é o seguinte:

```
1  class Node:
2    def __init__(self, data):
3      self.data = data
4      self.nextNode= None
5
6    def getData(self):
7      return self.data
8
9    def setData(self,data):
10     self.data=data
11
12   def getNextNode(self):
13     return self.nextNode
14
15   def setNextNode(self,newNode):
16     self.nextNode = newNode;
17
18 class List:
19   def __init__(self):
20     self.firstNode = None
21     self.lastNode = None
22
23   def isEmpty(self):
24     return self.firstNode is None
25
26   def insertAtBegin(self, value):
27     newNode = Node (value)
28     if self.isEmpty():
29       self.firstNode = self.lastNode = newNode
30     else:
31       newNode.setNextNode(self.firstNode)
32       self.firstNode = newNode
33
34 n= Node(1)
35 l= List()
```

Ao aplicarmos o nosso algoritmo de inferência a este código, vamos tratar as definições uma de cada vez, mas de modo semelhante.

Vamos centrar-nos na definição da classe `Node`. Como já foi referido, vamos inferir o tipo dos métodos da classe num novo contexto. O primeiro método a inferir o tipo é o método `__init__`. Como a definição do método não é mais do que a definição de uma função, vamos inferir o tipo do método `__init__`, como se este fosse uma função. No entanto, o argumento `self` é ignorado, como já vimos. Assim, o método é tratado como uma função que apenas tem um argumento - `data`. Este método contém duas atribuições. Na primeira atribuição a variável `self.data`, que ainda não existia no contexto, fica com o tipo do argumento `data` (σ) e é adicionada ao contexto. Como a variável é um atributo da classe, uma vez que é da forma `self.*`, é também adicionada ao contexto local da classe. A atribuição seguinte faz com que a variável `self.nextNode` seja adicionada aos contextos, corrente e local à classe, com o tipo `eNone`. Nenhum tipo é retornado. Assim, o método tem tipo `eAll([`σ`],eArrow([`σ`], eNone))`. Este tipo é adicionado ao contexto local da classe. O tipo dos restantes métodos da classe é inferido de modo semelhante. No final, o contexto local da classe (vamos designá-lo por conN) contém todos os métodos e atributos da classe e correspondentes tipos: {`self.nextNode::None, self.data::`σ`, __init__:: eAll([`σ`],eArrow([`σ`], eNone)), getData::eAll([`σ`], eArrow ([], `σ`)), setData :: eAll([`σ`], eArrow([`σ`], eNone)), getNextNode::eArrow ([], eNone), setNextNode:: eAll([`σ_1`],eArrow([`σ_1`], eNone))`}.

Ao inferirmos o tipo da classe `List`, o contexto corrente vai conter o tipo inferido para a classe `Node`. Desta forma, é possível instanciar a classe `Node`, na definição da classe `List`. A inferência de tipo para os métodos `__init__` e `isEmpty` é bastante semelhante à inferência descrita no parágrafo anterior. A inferência de tipos do método `insertAtBegin` é um pouco diferente. Na primeira atribuição a variável `newNode` fica com tipo `eClass`, uma vez que é uma instância da classe `Node`. No teste do condicional, é invocado o método `isEmpty`, da própria classe. O tipo do retorno deste método é booleano. De seguida, no caso em que o teste do condi-

cional é verdadeiro, são feitas duas atribuições. Nestas atribuições, os atributos `self.firstNode` e `self.lastNode`, que tinham tipo eNone, passam a ter o tipo da variável `newNode`, uma vez que `eNone` <: `eClass`. Caso o teste do condicional seja falso, é invocado o método `setNextNode` que retorna `eNone`. De seguida é feita uma atribuição em que não ocorre conflito de tipos, e que por isso retorna também `eNone`. Desta forma, o contexto da classe `List` é: { `self.firstNode :: eClass(Node, conN), self.lastNode :: eClass(Node, conN), __init__::eArrow([], eNone), isEmpty::eArrow([],eBool), insertAtBegin :: eAll([`σ_2`],eArrow([`σ_2`],[]))`}.

Nas duas classes, sempre que inferimos o tipo de um método, adicionámos esse método ao dicionário `overl`, dizendo a que classe ele pertence, e qual o tipo que ele tem nessa classe. Assim, no final da inferência para as duas classes, temos que `overl = {` `setNextNode:[(Node, eAll ([`σ_1`], eArrow([`σ_1`], eNone)))], nextNode:[(Node, None)], lastNode: [(List, eClass(Node, conN))], insertAtBegin: [(List, eAll([`σ_2`], eArrow([`σ_2`],[])))], getNextNode: [(Node,eArrow ([], eNone)], firstNode:[(List,eClass(Node, conN))], isEmpty:[(List,eArrow([],eBool)], getData:[(Node, eAll([`σ`]), eArrow ([], `σ`))], data:[(Node, `σ`)], __init__: [(List, eArrow([],eNone)),(Node, eAll([`σ`],eArrow([`σ`], eNone)))],setData: [(Node, eAll([`σ`], eArrow([`σ`], eNone)))]}`.

Depois de definidas estas duas classes, para criamos, por exemplo, um nó com o inteiro 1, basta instanciar a classe `Node` com esse argumento, ou seja, `Node(1)`. Ao inferir o tipo para a instanciação desta classe é chamado o método `__init__`, que recebe o inteiro 1 como argumento. Deste modo, o atributo `self.data` passa a ter tipo inteiro. Assim, após a atribuição `n= Node(1)`, temos que `n:: eClass(Node, {self.nextNode:: None, self.data::eInt, __init__::eArrow([eInt], eNone), getData::eArrow([], eInt), setData:: eArrow([eInt],eNone)), getNextNode::eArrow([], eNone), setNextNode::eAll ([`σ_1`], eArrow([`σ_1`], eNone))})`.

A instanciação da classe `List` é semelhante à da classe `Node`, apenas não existem argumentos. Deste modo, a variável `l`, após a atribuição `l = List()` tem o mesmo tipo que a classe `List`, ou seja, `l:: eClass(List, { self.firstNode :: eClass(Node,`

conN), self.lastNode :: eClass(Node, conN), __init__::eArrow([], eNone), isEmpty::eArrow([],eBool), insertAtBegin::eAll([σ_2],eArrow([σ_2],[]))}).

Capítulo 6

Conclusão

Um sistema de tipos permite prevenir a ocorrência de determinados erros durante a execução do programa. Para tal, define um conjunto de regras que associam tipos aos construtores do programa. Nas linguagens em que os tipos não são declarados explicitamente, mas para as quais existe um sistema de tipos associado, os tipos poderão ser obtidos por um processo de inferência. Neste trabalho, apresentamos um sistema de tipos, e um sistema de inferência de tipos, na ausência de execução, para o Python.

O Python é uma linguagem de muito alto nível, dinamicamente tipificada e que permite que as variáveis tenham diferentes tipos em diferentes locais do programa. Deste modo, para definir um sistema de inferência de tipos, na ausência de execução, como pretendíamos, reduzimos a linguagem Python a um subconjunto. O RPython foi o subconjunto escolhido.

O sistema implementado não funciona, ainda, para todo o RPython, mas para um subconjunto aceitável deste. Já são suportados os tipos predefinidos do Python, o uso de instruções condicionais, atribuições, operações e dos comandos `while` e `return`. É também permitida a definição e chamada a funções, a definição e instanciação de classes, e o acesso a métodos e atributos destas. Ainda não é suportada a importação de ficheiros, nem o uso de funções pré-definidas. O uso de iteradores ainda não está

implementado, assim como a instrução `for`. Um dos possíveis trabalhos futuros é a implementação das instruções em falta.

A integridade e completude do sistema de tipos definido não foi estudada. O estudo destas características implica a existência de uma semântica formal da linguagem Python, o que poderá ser um trabalho a desenvolver no futuro. A decidibilidade do algoritmo de inferência também não foi estudada.

Além de fazermos a inferência de tipos em Python, podíamos anotar o programa alvo da inferência, com os tipos inferidos. Assim, na importação de programas, se o programa que fosse importado já tivesse as anotações, não era necessário proceder à inferência de tipos. Deste modo, um programa que fosse muitas vezes importado, apenas seria alvo da inferência uma vez.

Actualmente a certificação de software, como correcto e seguro, é de extrema importância, especialmente para sistemas críticos e embebidos. Muitas das aplicações usadas nestes sistemas são desenvolvidas em linguagens de alto-nível, como o Python.

Ambicionamos encadear este sistema de inferência de tipos com a ferramenta de produção de obrigações de prova Why. O Why [Fil03] é uma ferramenta multi-linguagem e multi-demonstrador, podendo assim ser usada por programas anotados de diversas linguagens de programação e as obrigações de prova verificadas por diversos demonstradores automáticos ou assistentes de demonstração. A anotação de programas nas linguagens de programação fonte é feita utilizando uma versão das lógicas de Hoare [Hoa69]. Os programas anotados são traduzidos para uma linguagem funcional intermédia (HL) a partir da qual o Why produz as obrigações de prova. As obrigações de prova geradas, quando verificadas, provam a correcção e segurança do programa. Pretendemos traduzir a linguagem Python na linguagem HL. Ao contrário do Python, a linguagem HL é tipificada explicitamente. Recorrendo ao processo de inferência de tipos que desenvolvemos será possível anotar, com os tipos inferidos, os programas Python, o que ajudará na tradução entre as duas linguagens. Assim, o desenvolvimento deste sistema estático de inferência de tipos foi apenas o primeiro

passo para um projecto futuro que implemente a certificação estática de programas em Python.

Apêndice A

Sintaxe abstracta de um subconjunto do Python

{

mod = Module(stmt* body)

| Interactive(stmt* body)

| Expression(expr body)

−− not really an actual node but useful in Jython's typesystem.

| Suite(stmt* body)

stmt = FunctionDef(identifier name, arguments args, stmt* body, expr*decorator_list)

| ClassDef(identifier name, expr* bases, stmt* body, expr *decorator_ list)

|Return(expr? value)

| Assign(expr* targets, expr value)

−− not sure if bool is allowed, can always use int

−−use 'orelse' because else is a keyword in target languages

| While(expr test, stmt* body, stmt* orelse)

| If(expr test, stmt* body, stmt* orelse)

| Expr(expr value)

expr = BoolOp(boolop op, expr* values)

| BinOp(expr left, operator op, expr right)

| UnaryOp(unaryop op, expr operand)

| IfExp(expr test, expr body, expr orelse)

| Dict(expr* keys, expr* values)

| Compare(expr left, cmpop* ops, expr* comparators)

| Call(expr func, expr* args, keyword* keywords, expr? starargs, expr? kwargs)

| Num(object n) – a number as a PyObject.

| Str(string s) – need to specify raw, unicode, etc?

| Attribute(expr value, identifier attr, expr_ context ctx)

| Subscript(expr value, slice slice, expr_ context ctx)

| Name(identifier id, expr_ context ctx)

| List(expr* elts, expr_ context ctx)

| Tuple(expr* elts, expr_ context ctx)

expr_ context = Load | Store | Del | AugLoad | AugStore | Param

slice = Slice(expr? lower, expr? upper, expr? step)

| Index(expr value)

boolop = And | Or

operator = Add | Sub | Mult | Div | Mod | Pow | LShift | RShift | BitOr | BitXor
| BitAnd | FloorDiv

unaryop = Invert | Not | UAdd | USub

cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

arguments = (expr* args, identifier? vararg, identifier? kwarg, expr* defaults)

keyword = (identifier arg, expr value)

}

Referências

[Ayc04] John Aycock. Aggressive type inference. 2004.

[Can05] Brett Cannon. Localized type inference of atomic types in python, 2005.

[CW85] Luca Cardelli and Peter Wegner. *On Understanding Types, Data Abstraction, and Polymorphism*, volume 17. 1985.

[DM82] Luís Damas and Robin Milner. Principal type schemes for functional programs. In *ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*, pages 207–212, 1982.

[Fea09] Michael Furr and et al. Static type inference for ruby, 2009.

[Fil03] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

[Hin97] J. Roger Hindley. *Basic simple type theory.* Cambridge University Press, New York, NY, USA, 1997.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[Ped] Samuele Pedroni. Pypy. *Vancouver Python Workshop.*

[Pie02] Benjamin C. Pierce. *Types and programming languages.* 2002.

[Pro] PyPy Project. Pypy: flexible and fast python implementation.

[Py] Abstract syntax trees. *Python v2.6.5 documentation*.

[Rig04] Armin Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26, New York, NY, USA, 2004. ACM.

[Ros95] Guido van Rossum. Python reference manual. Technical report, Amsterdam, The Netherlands, 1995.

[RP06] Armin Rigo and Samuele Pedroni. Pypy's approach to virtual machine construction. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 944–953, 2006.

[RPH05] Armin Rigo, Samuele Pedroni, and Michael Hudson. Compiling dynamic language implementations. technical report d05.1. Technical report, 2005.

[Sal00] Michael Salib. Starkiller: a static type inferencer and compiler for python. *International Python Conference*, 2000.