UNIVERSIDADE TÉCNICA DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

SEMÂNTICA DA LINGUAGEM GNOME EM TyCO

ANTÓNIO MARIA ALARCÃO RAVARA
(LICENCIADO)

Dissertação para a obtenção do grau de mestre
em
Matemática Aplicada

Janeiro 1996

Resumo

Apresenta-se um fragmento da linguagem GNOME, de especificação formal de sistemas concorrentes, interactivos e reactivos. A linguagem é orientada a objectos e baseada em classes e em objectos. Uma especificação define uma comunidade de objectos, para a qual se pretende estabelecer uma semântica operacional. Para tal, introduz-se um cálculo de objectos concorrentes — o TyCO — com comunicação assíncrona e passagem de nomes, na linha do Cálculo-π, do Cálculo-ν e das Álgebras de Processos, como o CCS. Define-se o mapa de tradução da linguagem GNOME em TyCO. Uma classe GNOME é representada por um agente TyCO, sendo um objecto, instância da classe, representado por um processo, instância do agente. Apresenta-se ainda um sistema de tipificação para o TyCO, semelhante aos do Cálculo-λ, mas que atribui tipos aos nomes e tipificações (conjuntos de pares nome-tipo) aos processos. O sistema de tipos fornece especificação parcial, na forma de protocolos de comunicação para os processos. Mostram-se algumas das propriedades mais importantes do sistema de tipos, como a invariância de um tipo durante uma redução, a ausência de erros em tempo de execução nos processos tipificáveis, a computabilidade da inferência de tipos e de tipificações e a decidibilidade da atribuição de tipos e de tipificações. Discute-se como usar o sistema de tipos para fazer provas de correcção parcial de especificações face a um certo tipo de propriedades de invariância, concluindo-se que tal não é possível.

Abstract

A fragment of the language GNOME for formal specification of concurrent, interactive and reactive systems is presented. It is an object-oriented language, class-based and object-based. A specification defines a community of objects, to which will be established an operational semantics. Aiming that, an untyped calculus of concurrent objects — TyCO — with asynchronous communication and name-passing it's introduced. It is in the trends of the π-Calculus, the ν-Calculus and the Process Algebra's, like CCS. The semantic map of the language GNOME in TyCO is defined. A GNOME class is represented by a TyCO agent and an object, instance of a class, is represented by a process, instance of an agent. A typing system for TyCO is also presented, similar to those of the λ-Calculus, but assigning types to names and typings (sets of name-type pairs) to processes. The type system provide partial specification as communication protocols for processes. Some of the most important properties of type system are shown, such as the subject-reduction, the absence of run-time errors in typable processes, the computability of the inference of types and typings and the decidability of the assigning of types and typings. It is discussed how to use the type system to prove the partial-correctness of specifications regarding a certain type of invariance properties, concluding that it is not possible.

Palavras Chave / Keywords

Cálculo de Objectos Concorrentes	Calculus of Concurrent Objects
Comunicação	Communication
Especificação	Invariance Properties
Propriedades de Invariância	Object-Oriented Systems
Semântica Operacional	Operacional Semantics
Sistemas de Inferência de Tipos	Type Inference Systems
Sistemas Orientados a Objectos	Specification

Agradecimentos

O trabalho que apresento nesta dissertação é fruto de um longo processo onde os contributos de muitas pessoas foram fundamentais.

A exigência, encorajamento e entusiasmo do Professor Amílcar Sernadas, meu orientador, foram factores decisivos.

A permanente disponibilidade e grande interesse que sempre manifestou o Doutor Vasco Vasconcelos, co-orientador desta dissertação, seu crítico e inspirador, foram inexcedíveis.

Saliento o apoio que foi o excelente ambiente de trabalho no grupo de Lógica e Computação do Departamento de Matemática. Destaco especialmente a Professora Cristina Sernadas, que marca o espírito deste grupo, e agradeço ao Pedro e ao Jaime, que leram partes desta dissertação e deram valiosas sugestões, ao Carlos, pelas discussões várias e à Paula e ao João, pela muita ajuda com o LaTeX.

Incomensoravelmente importante foi o encorajamento e interesse de tantas amigas e de tantos amigos, que me foram apoiando e incentivando ao longo deste processo. Em particular, que leu parte desta dissertação.

Fica um especial agradecimento à minha familia, com relevo para a Dina.

Indice

Capítulo 1

Introdução

A concepção e o desenvolvimento de sistemas de informação é, na engenharia informática, uma área em clara expansão, o que corresponde à implantação de facto da informática no quotidiano das comunidades.

Estes sistemas são de elevada complexidade, tendo a sua programação um conjunto de exigências relativamente às quais não há ainda respostas e soluções globalmente aceites.

Vários autores fazem o diagnóstico da situação e propõem abordagens. Como se salienta em [Sernadas *et al* 92a], apesar dos progressos extremamente rápidos ao nível do desenvolvimento de equipamento ("hardware"), a área de produção de aplicações ("software") tem evoluído muito mais lentamente, falando-se mesmo de uma "crise", que se arrasta desde os anos sessenta, nomeadamente quanto à produtividade, custo e qualidade. Por isso alguns autores defendem a adopção de uma única linguagem em todas as fases do desenvolvimento de um sistema [Nierstrasz 89, Sernadas e Sernadas 94], de forma a reduzir as tarefas de manutenção e as descontinuidades entre as diferentes etapas do ciclo de vida de uma aplicação, equilibrar a versatibilidade e a expressibilidade das linguagens e facilitar o trabalho cooperativo.

Em Ciência da Computação muitos esforços se têm centrado nas estratégias e metodologias de desenvolvimento destes sistemas que são concorrentes, interactivos e reactivos. São cada vez mais aceites as abordagens formais, com linguagens de especificação (em complemento das de programação tradicionais), centradas em objectos, abordagens que são já novos paradigmas, tratados por inúmeros autores (ver por exemplo [Wegner 90]). Visam fornecer um instrumento que seja aplicável em todas as fases do ciclo de vida de uma aplicação, desde a análise à manutensão e re-utilização.

Uma possível falta de flexibilidade das linguagens é compensada pelo rigor da semântica e por permitir definir um sistema dedutivo para provas de correcção das especificações face a propriedades de segurança e de animação [Sernadas e Sernadas 94].

São exemplos destas novas abordagens ao desenvolvimento de sistemas de informação as linguagens OBLOG [Sernadas *et al* 92b, ESDI 93] e TROLL [Jungclaus *et al* 91]. Uma versão textual e revista da abordagem OBLOG é a linguagem GNOME, proposta por Sernadas e Ramos em [Sernadas e Ramos 94, Ramos e Sernadas 95a]. Nesta dissertação pretende-se definir uma semântica operacional para a linguagem GNOME, baseada no Cálculo de Objectos Concorrentes TyCO (Typed Concurrent Objects), proposto por Vasconcelos em [Vasconcelos 94d, Vasconcelos 94c].

A linguagem GNOME é orientada a objectos e destina-se à especificação de sistemas concorrentes, reactivos e interactivos. Baseia-se em classes e em objectos, sendo as classes moldes para a criação de objectos e estes unidades de capsulação. Suporta concorrência entre objectos animados, tem mecanismos de herança (liberal e estrita), e tem ainda bibliotecas e unidades de trabalho cooperativo.

O cálculo TyCO de objectos concorrentes suporta comunicação assíncrona entre objectos, através de mensagens que transportam nomes. É inspirado em modelos de concorrência bem conhecidos, como o Cálculo-π [Milner 91, Milner *et al* 92], o Cálculo-ν de Honda [Honda e Tokoro 91], o Modelo de Computação de Actores de Hewitt [Agha 86, Hewitt *et al* 73] ou as Álgebras de Processos, de que são exemplos o CCS [Milner 89] e o CSP [Hoare 85].

O TyCO é implicitamente tipificado, não havendo informação de tipos explícita nos termos, mas esta pode ser inferida através de sistemas de tipos. Estes sistemas estão na linha dos do Cálculo-λ [Barendregt 92, Hindley e Seldin 86], mas atribuem tipos aos nomes e tipificações (conjuntos de pares nome-tipo) aos processos.

No capítulo 2 apresenta-se a linguagem GNOME: fala-se das suas características enquanto linguagem de especificação orientada a objectos, define-se a gramática do fragmento da linguagem com que se trabalha, recorrendo à notação EBNF[1], e ilustra-se a sua semântica informalmente, através de exemplos.

No capítulo 3 apresenta-se o TyCO: define-se a sua linguagem, os termos, as fórmulas e o cálculo, que estabelece o mecanismo de computação — a comunicação — que constitui um esquema de redução. Mostra-se como codificar tipos de dados e definem-se alguns processos que enriquecem a sintaxe dos termos e serão úteis para simplificar a codificação de classes GNOME em TyCO. Este "açúcar sintático" é recolhido no apêndice B.

No capítulo 4 desenvolve-se a semântica do GNOME em TyCO: começa-se por construir termos que representam classes muito simples e vão-se introduzindo gradualmente construtores GNOME mais complexos, que, pela sua semântica, vão modelando a forma dos termos TyCO que representam essas classes, pois pretende-se que os termos TyCO obtidos tenham um comportamento equivalente às classes GNOME. Apresenta-se, no final, um termo que representa uma classe que contém, no fundamental, as primitivas do fragmento da linguagem que se considerou neste trabalho. No apêndice A desenvolve-se o termo que representa uma classe com estrutura típica, considerando o fragmento da linguagem GNOME que se trata.

No capítulo 5 apresenta-se um sistema de inferência de tipos que permite tipificar os processos TyCO. Provam-se algumas das suas mais importantes propriedades, como a invariância do tipo sob a redução, a não ocorrência de erros em tempo de execução nos processos tipificáveis, a computabilidade da inferência de tipos e tipificações e a decidibilidade da sua atribuição. No final, discute-se a possibilidade de usar o sistema para provar a

[1] Extended Backus Naur Form

correcção parcial das especificações face a certas propriedades de invariância, propondo-se uma abordagem e aplicando-a a exemplos.

No capítulo 6 resumem-se os resultados de cada capítulo, apontam-se perspectivas para trabalho futuro neste assunto e referem-se outras semânticas para a linguagem GNOME e outras linhas de investigação na área das lógicas de especificação de objectos.

Capítulo 2

A linguagem GNOME: descrição informal e gramática

2.1 Apresentação da linguagem

A linguagem GNOME é, segundo [Sernadas e Ramos 94], uma *linguagem orientada a objectos, de especificação formal de sistemas concorrentes, reactivos e interactivos*. Está baseada em classes e em objectos, sendo as classes moldes para a criação de objectos similares (instâncias das classes), e sendo os objectos unidades de capsulação. A linguagem é fortemente tipificada, sendo as classes tipos e podendo-se introduzir subclasses, estando assim disponíveis mecanismos de herança, quer estrita, quer liberal.

Os objectos em GNOME são entidades dinâmicas, caracterizadas por um estado interno, definido por um conjunto de atributos, que pode ser alterado, quer por iniciativa própria — têm animação, podendo executar acções internas — quer por interacção com outras entidades — executam acções externas, também chamadas serviços, a pedido do exterior. O comportamento de um objecto é dependente do seu estado, caracterizando-se um sistema por ser uma comunidade de objectos concorrentes, que podem interactuar (caracterização comum de objecto — ver [Sernadas *et al* 92a, Wegner 90]). A criação e morte de objectos é dinâmica, sendo no entanto um objecto persistente, por omissão.

Logo, a linguagem suporta concorrência entre objectos animados, que podem interagir uns com os outros ou com um utilizador, ou por comunicação síncrona (por invocação de serviços, pedidos do exterior), ou por comunicação assíncrona (por emissão de eventos).

A linguagem GNOME é textual e de muito alto nível, tendo um conjunto fixo de tipos de dados base e dispondo de construtores de tipos, tendo instanciação de bibliotecas paramétricas e unidades de trabalho cooperativo (as regiões). É compatível com implementações distribuidas.

Em [Sernadas e Ramos 94] é estabelecida para a linguagem GNOME uma semântica operacional rigorosa e uma semântica axiomática baseada em lógica temporal.

Em [Ramos 94, Ramos e Sernadas 95b] é desenvolvida outra axiomatização para a linguagem GNOME. Baseia-se em GCL (GNOME *Certification Logic*), que é uma extensão da lógica de especificação de objectos OSL (*Object Specification Logic*), proposta em [Sernadas e Sernadas 94, Sernadas *et al* 95].

Foi desenvolvido um compilador da linguagem, que abrange um fragmento bastante alargado desta, e que gera código C++ [Carmo e Penedo 94].

Começa-se por apresentar a sintaxe da linguagem, através da definição da sua gramática. Depois explica-se informalmente a semântica e ilustra-se com alguns exemplos.

2.2 Sintaxe do GNOME

Neste trabalho considera-se um fragmento da linguagem que não contém algumas primitivas, como a emissão de eventos, as reacções, as regiões, as bibliotecas e a herança. O núcleo que se vai tratar permite colocar em evidência algumas das principais características desta linguagem, sendo adequado para uma primeira abordagem à semântica do GNOME em TyCO. As primitivas que ficaram de fora, sendo potencialidades importantes da linguagem, não são fundamentais para salientar as suas funcionalidades básicas (aqui consideradas), como a existência de classes, a criação e morte de objectos, a sincronização, a concorrência e a existência de não determinismo na execução das acções internas e na escolha entre as alternativas (activadas) das acções.

Os termos TyCO encontrados para expressar a semântica do GNOME são já de razoável complexidade e, como se verá, expressam bem a semântica das características referidas.

Tratar as primitivas que ficaram de fora pode ser fácil, como no caso dos eventos (por serem um mecanismo de comunicação assíncrona e ser esta também a forma que assume a comunicação em TyCO), ou mais complexo, como no caso da herança.

Na apresentação da gramática da linguagem convenciona-se que as palavras reservadas surgirão em estilo "typewriter" e serão símbolos terminais da gramática; em estilo carregado surgirão símbolos não terminais e em estilo itálico surgirão alguns desses símbolos não terminais, dos quais não será detalhada a sintaxe, tais como os identificadores e os construtores de tipos.

Definição 2.2.1 *Gramática da linguagem* GNOME

unidade ::=`spec` *id_espec*
  dec_tipo*
  classe*
  `end spec`

dec_tipo ::= `type` *id_dec_tipo* `=` tipo

tipo ::= tipo_prim | const_tipo | *id_dec_tipo* | *id_classe*

tipo_prim ::= `bool` | `nat`

const_tipo ::= *enumeração* | *estrutura*

classe ::= `class` *id_classe*
`interface` **serviço***
`body` **corpo**
`end` *id_classe*

serviço ::= [`cat`] `serv` *id_serv*
parâmetro*
resultado*

cat ::= `birth` | `death`

parâmetro ::= `par` *id_par*: **tipo**

resultado ::= `res` *id_res*: **tipo**

corpo ::= (`slot` *id_atributo*: **tipo**)*
(`serv` *id_serv* **método**)*
([`death`] `act` *id_acção* **método**)*

método ::= **método** | `alt` *id_alt* **método**

método ::= **activação***
chamada*
valoração*
resposta*

activação ::= `enb` *condição*

chamada ::= `call` *id_serv* [`as` *id_serv*] `of` *expressão*:*id_classe*
(`arg` *id_serv.id_par* `=` *expressão*)*

valoração ::= `val` *id_atributo* $\ll$ *expressão*

resposta ::= `ret` *id_serv.id_res* `=` *expressão* ∎

Como se sabe, uma gramática não define completamente uma linguagem. No entanto, as produções apresentadas, junto com as observações seguintes, são suficientes para compreender as especificações que se seguem para efeitos de ilustração, as quais são devidamente comentadas.

Observação 2.2.2 *Algumas clarificações sobre a gramática*

- todas as construções que envolvem parâmetros têm dependências de contexto; importa atender à compatibilidade de tipos de dados nas expressões, para que estas tenham sentido;

- na produção **chamada**, a expressão deve identificar uma instância existente da classe *id_classe* (que não a própria); esta expressão pode ser simplesmente o atributo especial `next` (que se pode ler mas não se pode escrever), que denota a identidade da próxima instância a ser criada — este atributo só pode surgir na invocação de um serviço de nascimento ou em cláusulas de valoraçao, passagem de argumentos ou retorno de valores, devendo, nestes casos, ser prefixado com o nome da classe a que se refere;

- quando se quer identificar a própria instância em que se trabalha usa-se o atributo especial `self` (tal como `next`, pode-se ler mas não se pode escrever), que denota essa instância. ■

2.3 Breve descrição da semântica

Uma especificação define uma comunidade de objectos, através da declaração de um conjunto de classes, das quais os objectos são instâncias.

As classes são constituidas por uma parte visível, chamada a *interface*, e uma parte capsulada, chamada o *corpo*. Na interface declaram-se os serviços (acções públicas), que têm dois tipos de argumentos: de entrada, chamados parâmetros, e de saída, chamados resultados. No corpo estão os detalhes da implementação do comportamento do objecto: declara-se o conjunto dos seus atributos e os métodos de cada uma das suas acções.

A execução de uma acção é transacional — executa-se atomicamente, sem possibilidade de intercalação. Cada objecto tem comportamento "sequencial" — só há paralelismo entre objectos diferentes.

Os métodos podem ser simples ou com alternativas. Um método com alternativas escolhe de forma não determinista, de entre as activadas, uma para executar, exigindo-se justiça forte [Ramos 94]. Um método simples pode ter quatro tipos de cláusulas: condições de activação, invocações de serviços, valorações de atributos e retornos de valores (só disponível para acções externas).

Para executar um método (ou uma alternativa de um método) avaliam-se primeiro as suas condições de activação. Se estas forem verdadeiras fazem-se as eventuais chamadas de serviços, verificando-se se cada um deles está, por sua vez, activado, pois só em caso afirmativo estará o método activado. Depois, caso se tenha verificado que está activado, realizam-se as atribuições de valores aos atributos e retornam-se os resultados. Se as suas condições de activação são falsas, não está activado e não pode ser executado. Igualmente não está activado se algum dos serviços chamados o não estiver ou a instância invocada não existir.

Se o método tiver alternativas, a avaliação da activação do método é a avaliação da activação de alguma das suas alternativas, aplicando-se o que se acabou de descrever.

2.4 Exemplos

Apresentam-se algumas especificações, propostas em [Sernadas e Ramos 94], para ilustrar a semântica do GNOME. O primeiro exemplo tem duas classes, em interacção.

Exemplo 2.4.1 *Semáforos*

`spec` semaforos

  `class` Sem

    `interface`

      `birth serv` *new*

```
      serv req
      serv rel
    body
      slot s : nat
      serv new
        val s ≪ 1
      serv req
        enb s > 0
        val s ≪ s − 1
      serv rel
        enb s < 1
        val s ≪ s + 1
  end Sem
  class Cliente
    interface
      birth serv new
        par s : Sem
      serv noncrit
      serv request
      serv crit
      serv release
    body
      slot st : [nc,rq,cr,rl]
      slot s : Sem
      serv new
        val st ≪ nc
        val s ≪ new.s
      serv noncrit
        enb st = nc
        val st ≪ rq
      serv request
        enb st = rq
        call req of s : Sem
        val st ≪ cr
      serv crit
        enb st = cr
        val st ≪ rl
      serv release
        enb st = rl
        call rel of s : Sem
        val st ≪ nc
  end Cliente
end spec
```
■

Quando é criada uma instância da classe Cliente (por invocação do seu serviço *new*) o atributo de estado (st) é inicializado a `nc` e o atributo s é inicializado com a identidade de uma instância da classe Sem (que lhe é passada no parâmetro do serviço).

Quando depois é chamado um dos seus serviços (por exemplo, *request*) é feita a avaliação da condição de activação, verificando-se se o valor do atributo de estado permite realizar essa acção; se sim (eventualmente) invoca-se um serviço da instância de Sem que está identificada no parâmetro, que, se estiver activado, altera o valor do seu atributo, alterando-se depois o valor do atributo de estado; se não estiver activado, o serviço invocador também não o estará.

Além da interacção, outros mecanismos importantes em Gnome são a animação interna dos objectos (por execução de acções de iniciativa própria — fornece iteratividade) e a criação e morte de instâncias de forma dinâmica (fornece recursividade).

Exemplifica-se com duas versões da especificação da função factorial: uma iterativa e outra recursiva.

Exemplo 2.4.2 *Factorial iterativo*

```
class Fact
 interface
  birth serv req
   par n : nat
  death serv ans
   res n! : nat
 body
  slot n : nat
  slot n! : Fact
  serv req
   val n ≪ req.n
   val n! ≪ 1
  serv ans
    enb n = 1
    ret ans.n! = n!
  act passo
   enb (n > 1)
   val n! ≪ n * n!
   val n ≪ n − 1
end Fact
```
■

Ao ser criada uma instância da classe Fact (por invocação do seu serviço *req*) é-lhe passado o valor de que se quer calcular o factorial, que é guardado no atributo n; o atributo $n!$ é inicializado a 1.

Se n é maior que 1, desencadeia-se a animação interna da instância, sendo executada a acção passo, que faz as multiplicações de forma a calcular o factorial do valor passado.

O invocador da instância da classe Fact pede depois o resultado, chamando então o serviço *ans* (que só está activado quando o atributo n tiver o valor 1).

Exemplo 2.4.3 *Factorial recursivo*

```
class Fact
 interface
  birth death serv calc
```

```
    par n : nat
    res n! : nat
  body
    serv calc
      alt triv
        enb calc.n < 2
        ret calc.n! = 1
      alt non - triv
        enb calc.n > 1
        call calc as calc2 of next : Fact
        arg calc2.n = calc.n - 1
        ret calc.n! = n * calc2.n!
end Fact
```

∎

Nesta proposta, quando é criada uma instância da classe Fact (por invocação do seu serviço *calc*), o seu parâmetro n recebe o valor de que se quer calcular o factorial; se esse valor for menor que 2, a instância devolve o valor 1 (resultado da função) e morre; se for maior que 1 vai criar outra instância, passando-lhe o valor n-1 e, por sua vez, esta nova instância criará outra, sucessivamente até que o valor passado não seja superior a 1; depois, os resultados dos serviços invocados vão sendo retornados às instâncias, obtendo-se finalmente o valor do factorial, que é passado ao invocador.

Capítulo 3

TyCO: um cálculo para objectos concorrentes

3.1 A linguagem TyCO

3.1.1 Fundamentos

O **Ty**ped **C**oncurrent **O**bjects (TyCO) é, segundo [Vasconcelos 94c], um *cálculo de objectos concorrentes, com comunicação assíncrona por mensagens que transportam nomes.*

Foi proposto por Vasconcelos, sendo inspirado no Cálculo-π de Milner, Parrow e Walker e no Cálculo-ν de Honda. Recolhe ainda elementos no Modelo de Computação de Actores de Hewitt e nas Álgebras de Processos, como o **C**alculus of **C**ommunicating **S**ystems (CCS) e o **C**ommunicating **S**equencial **P**rocesses (CSP).

Este cálculo visa fornecer uma *semântica operacional e sistemas de inferência de tipos*[1] para linguagens de programação concorrentes e orientadas a objectos. O seu mecanismo de computação baseia-se num esquema de redução[2] e designa-se por *comunicação*.

O cálculo é implicitamente tipificado — não há informação de tipos explícita nos termos, mas esta pode ser inferida a partir de sistemas de atribuição de tipos aos nomes e de tipificações (conjuntos de pares nome-tipo) aos processos. Estes sistemas garantem que os processos tipificáveis não encontram erros em tempo de execução, e têm outras importantes propriedades, que se discutem com algum detalhe no capítulo 5.

Neste capítulo apresenta-se primeiro a linguagem TyCO: define-se a sua sintaxe, faz-se a codificação dos tipos de dados — valores e operações dos booleanos e naturais — mostrando-se também como se faz a avaliação de expressões, e definem-se ainda construtores derivados, que serão úteis na apresentação da semântica da linguagem GNOME em TyCO[3].

Depois apresenta-se o cálculo — as fórmulas e a relação de redução — que fornece a semântica operacional, à custa do conceito de comunicação, ilustrando a sua aplicação

[1] Apresentam-se estes sistemas no capítulo 5.

[2] Semelhante ao do Cálculo-λ [Barendregt 74, Hindley e Seldin 86]

[3] Ver o próximo capítulo.

com alguns dos construtores apresentados antes.

3.1.2 Sintaxe

Define-se a linguagem com um alfabeto e com a noção de termo.

Definição 3.1.1 Um *alfabeto* da linguagem TyCO é constituído por:

- Um conjunto **NP**, numerável, cujos elementos se chamam **nomes de processos**[4];
- Um conjunto **L**, cujos elementos se chamam **etiquetas**;
- Um conjunto **NA**, numerável, cujos elementos se chamam **nomes de agentes**[5];
- Os símbolos $\lhd$ $\rhd$ [ ] & **,** ν ! : ( )
- O símbolo de redução, $\rightarrow$, e o símbolo de equivalência, $\equiv$ ∎

Definição 3.1.2 Os conjuntos dos *termos de processo* e dos *termos de agente*, respectivamente **TP** e **TA**, são definidos simultânea e indutivamente do modo que se segue:

- $(\tilde{x})P \in \mathbf{TA}$ desde que $\tilde{x} \in \mathbf{NP}^*$ e $P \in \mathbf{TP}$. Considera-se que em $\tilde{x}$ não há repetição de nomes;
- $\texttt{rec}\, X.A \in \mathbf{TA}$ desde que $X \in \mathbf{NA}$ e $A \in \mathbf{TA}$;
- $\texttt{let}\ X\ \texttt{=}\ A\ \texttt{in}\ P \in \mathbf{TP}$ desde que $X \in \mathbf{NA}$, $A \in \mathbf{TA}$ e $P \in \mathbf{TP}$;
- $a \lhd l : \tilde{v} \in \mathbf{TP}$ desde que $a \in \mathbf{NP}, l \in \mathbf{L}$ e $\tilde{v} \in \mathbf{NP}^*$;
- $a \rhd [l_1 : A_1\ \& \cdots \&\ l_n : A_n] \in \mathbf{TP}$ desde que $a \in \mathbf{NP}$, $l_i \in \mathbf{L}$ e $A_i \in \mathbf{TA}$. Considera-se que $l_i \neq l_j$, $\forall\, i, j \colon 1 \leq i, j \leq n$;
- $P, Q \in \mathbf{TP}$ desde que $P \in \mathbf{TP}$ e $Q \in \mathbf{TP}$;
- $\nu x P \in \mathbf{TP}$ desde que $x \in \mathbf{NP}$ e $P \in \mathbf{TP}$;
- $A(\tilde{v}) \in \mathbf{TP}$ desde que $A \in \mathbf{TA}$ e $\tilde{v} \in \mathbf{NP}^*$;
- $X(\tilde{v}) \in \mathbf{TP}$ desde que $X \in \mathbf{NA}$ e $\tilde{v} \in \mathbf{NP}^*$. ∎

Observação 3.1.3

- Os elementos de **NP** e de **NA** têm, na linguagem, a função de *variáveis*;
- Num contexto informal, os termos de processo serão simplesmente designados por *processos*. Da mesma forma, os termos de agente serão designados por *agentes*;
- Os processos da forma $a \rhd M$ serão designados por **objectos**, sendo a um nome de processo, dito o identificador ou *nome* do objecto e M uma colecção (não ordenada) de serviços. Um serviço tem a forma $l{:}A$, sendo l a sua etiqueta e A o seu método;

[4] Em [Vasconcelos 94c] são referidos como *nomes*, designando-se o conjunto a que pertencem por **N**.
[5] Em [Vasconcelos 94d] são introduzidos com a designação de *variáveis de agentes*.

- Os processos da forma $a \lhd l : \tilde{v}$ serão designados por **mensagens etiquetadas assíncronas**, onde a é um nome de processo, que se diz o *alvo* da mensagem (identifica o objecto ao qual se destina a mensagem), l é a etiqueta que vai seleccionar um serviço no objecto alvo, e $\tilde{v}$ é o conteúdo da mensagem (a sequência de nomes a passar), dizendo-se que $l{:}\,\tilde{v}$ é a *comunicação*;

- P, Q denota a **composição concorrente** de processos — é o processo que resulta de pôr os processos P e Q a correr em paralelo, podendo estes comunicar entre si ou agir independentemente;

- $\nu x P$ denota a **restrição de âmbito** de um nome de processo x a um termo de processo P — é um processo onde x é local a P (só é "visível" no âmbito determinado por P);

- Os termos de agente[6] permitem definir **famílias de processos parametrizados**.

  Sendo A um termo de agente, $A(\tilde{v})$ é o processo que denota a aplicação de uma sequência de nomes de processos $\tilde{v}$ a A. Se A for da forma $(\tilde{x})P$, então $A(\tilde{v})$ denota o processo P onde os nomes $\tilde{v}$ substituiem os nomes $\tilde{x}$, seus parâmetros (sequência de nomes);

- A construção `rec` permite definir termos de agente recursivos.

- A construção `let` permite definir **processos locais** — declara-se um nome de agente que vai ser depois usado num processo que é o "resto do programa".

  Em `let` X `=` A `in` P, as ocorrências de $X(\tilde{v})$ em P denotam os processos $A(\tilde{v})$. ■

Um objecto pode ser referido por vários nomes e, igualmente, um nome (de processo) pode estar associado a varios objectos. Mas nem todos os nomes de processos estão necessariamente associados a objectos — uma mensagem para um nome é um termo bem formado, mesmo que esse nome não seja referência de nenhum objecto.

Observação 3.1.4 Para simplificação de notação podem-se omitir parênteses, considerando-se a associatividade à esquerda. Considera-se também que o operador de restrição de âmbito ("ν") tem precedência sobre o operador de composição concorrente (",").

Vão ser então assumidas as seguintes *abreviaturas*:

- considerando-se a associatividade à esquerda, $\nu x_1 \cdots \nu x_n P$ pode ser abreviado para $\nu x_1 \cdots x_n P$;

- os objectos e as mensagens são os termos de processo básicos; denota-se por $\tilde{P}$ uma composição concorrente de termos básicos;

- $a \lhd l : \varepsilon$ é a mensagem vazia, que pode ser abreviada para $a \lhd l$;

- $a \rhd [l : (\varepsilon)P]$ é o objecto com um serviço sem parâmetros, que pode ser abreviado para $a \rhd [l : P]$;

[6] Em [Vasconcelos 94d] são referidos como *agentes*

- $a \triangleright []$ é um objecto inanimado. A partir dele pode-se definir um processo que não faz nada — $\nu x\; x \triangleright []$ — que se denota por $\mathbf{0}$ e se chama *inacção*.

  Por exemplo, $f \triangleright [stop{:}\mathbf{0}]$ é um processo que termina uma computação quando recebe uma mensagem $f \lhd stop$;

- quando P for o "resto do programa" abrevia-se `let` X `=` A `in` P por X `=` A. Igualmente se pode escrever `rec` X `=` A como abreviatura de X `=` `rec` $X.A$. ∎

Apresentada a sintaxe importa ver como se a usa enquanto linguagem de especificação. Codificam-se primeiro os tipos de dados disponíveis em GNOME[7] e depois definem-se alguns construtores derivados (à custa dos termos), que constituirão "açúcar" sintáctico da linguagem.

3.1.3 Codificação dos tipos de dados

Definição 3.1.5 Os valores booleanos podem ser implementados definindo objectos (instâncias de agentes) que respondem a uma mensagem *val* emitindo uma mensagem *true* ou *false* para um dado nome passado na mensagem *val*:

$$\mathrm{True} \;=\; (b)\, b \triangleright [\mathit{val} : (r)\, r \lhd \mathit{true}]$$

$$\mathrm{False} \;=\; (b)\, b \triangleright [\mathit{val} : (r)\, r \lhd \mathit{false}]$$

Nesta proposta os valores são efémeros (quando recebem uma mensagem *val* são "consumidos") — outra possível codificação dos valores booleanos é definindo agentes, cujas instâncias sejam objectos persistentes, com comportamento semelhante aos anteriores, como por exemplo:

$$\texttt{rec}\ \mathrm{True} \;=\; (b)\, b \triangleright [\mathit{val} : (r)\, r \lhd \mathit{true},\ \mathrm{True}(b)]$$

$$\texttt{rec}\ \mathrm{False} \;=\; (b)\, b \triangleright [\mathit{val} : (r)\, r \lhd \mathit{false},\ \mathrm{False}(b)]$$

Existindo no sistema os objectos persistentes $\mathrm{True}(t)$ e $\mathrm{False}(f)$, em paralelo, as operações são serviços do agente seguinte:

$$\begin{array}{l}
\texttt{rec}\ \mathrm{Boolops} \;=\; (b)\, b \triangleright [\, \mathit{not} : (v\ r)\ \nu x\ v \lhd \mathit{val} : x, \\
\qquad\qquad\qquad\qquad\qquad x \triangleright [\mathit{true} : r \lhd \mathit{false}, \mathrm{Boolops}(b)\ \& \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{false} : r \lhd \mathit{true}, \mathrm{Boolops}(b)]\ \& \\
\qquad\qquad\qquad \mathit{or} : (u\ v\ r)\ \nu x\ u \lhd \mathit{val} : x, \\
\qquad\qquad\qquad\qquad\qquad x \triangleright [\mathit{true} : r \lhd \mathit{true}, \mathrm{Boolops}(b)\ \& \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{false} : v \lhd \mathit{val} : x, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad x \triangleright [\mathit{true} : r \lhd \mathit{true}, \mathrm{Boolops}(b)\ \& \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{false} : r \lhd \mathit{false}, \mathrm{Boolops}(b)]]\ \& \\
\qquad\qquad\qquad \cdots\,]
\end{array}$$

∎

[7] seguindo [Vasconcelos 94c]

Definição 3.1.6 À custa de dois agentes, Zero e Suc, apresentados a seguir, podem ser construidos os números naturais:

$$\texttt{rec}\ \mathrm{Zero} \ = \ (\mathit{self})\ \mathit{self} \rhd [\mathit{val} : (r)\ r \lhd \mathit{zero}, \mathrm{Zero}(\mathit{self})]$$

$$\texttt{rec}\ \mathrm{Suc} \ = \ (\mathit{self}\, n)\ \mathit{self} \rhd [\mathit{val} : (r)\ r \lhd \mathit{suc} : n, \mathrm{Suc}(\mathit{self}\, n)]$$

Uma instância de Zero, responde a um pedido de avaliação (uma mensagem com etiqueta *val*), com uma mensagem *zero* para o nome que lhe foi passado. Uma instância de Suc responde que é o sucessor de determinado nome de processo (que representará o natural predecessor).

Com estes dois construtores podem-se definir os naturais. As operações que se desejar implementar sobre os naturais podem ser serviços de um agente (a quem se pode chamar Natop, por exemplo), tal como se fez para os booleanos. ■

Exemplo 3.1.7 *Um valor natural*
O natural "1" é implementado como se segue:

$$\mathrm{Zero}(z), \mathrm{Suc}(1, z)$$

De forma semelhante se implementam todos os outros naturais. ■

3.1.4 Construtores derivados

Quando se programa em TyCO são frequentes algumas construções típicas, que se podem definir como construtores derivados, enriquecendo a sintaxe base da linguagem.

Alternativas

Em programação são fundamentais as alternativas. Para as implementar é necessário tratar também as expressões booleanas.

Definição 3.1.8 *Alternativas simples*
O processo seguinte representa a **alternativa simples**, tendo como guarda a condição $op(\tilde{x})$, onde op é uma (e uma só) operação e $\tilde{x}$ os seus argumentos, sendo o resultado da sua avaliação booleano:

$$\texttt{if}\ op \lhd op : \tilde{x}\ \texttt{then}\ A_1\ \texttt{else}\ A_2\ \texttt{fi} \stackrel{\mathrm{def}}{=} \nu y\ op \lhd op : \tilde{x}y, y \rhd [\mathit{true} : A_1\ \&\ \mathit{false} : A_2]$$

onde y não ocorre livre[8] em A_1 ou A_2 e op é uma instância do agente que disponibiliza a operação op envolvida na guarda.

Se a condição é avaliada verdadeira, resulta a mensagem $y \lhd \texttt{true} : \tilde{v}$ ($\tilde{v}$ são os eventuais resultados), que interage com o objecto localizado em y e lança o processo $A_1(\tilde{v})$; caso contrário, é lançado o processo $A_2(\tilde{v})$.

A computação descrita é uma forma de redução, assente na comunicação[9].

[8] Ver definição 3.2.1.
[9] Ver definições 3.2.2 e 3.2.4.

Quando a guarda é simplesmente um valor booleano, representado pelo objecto b, instância de um dos agentes apresentados em 3.1.6, a alternativa passa a ser:

$$\texttt{if}\ b\ \texttt{then}\ A_1\ \texttt{else}\ A_2\ \texttt{fi} \stackrel{\text{def}}{=} \nu x\ b \lhd \mathit{val} : x, x \rhd [\mathit{true} : A_1\ \&\ \mathit{false} : A_2]$$

∎

Definição 3.1.9 *Alternativas generalizadas*
Pode-se também definir um processo que representa a **alternativa generalizada**:

$$\begin{array}{ll} \texttt{case}\ s \lhd op : \tilde{x}\ \texttt{of} & \\ \quad l_1 : A_1 & \\ \quad \vdots & \\ \quad l_n : A_n & \\ \texttt{esac} & \stackrel{\text{def}}{=} \nu y\ s \lhd op : \tilde{x}y, y \rhd [l_1 : A_1\ \& \cdots \&\ l_n : A_n]) \end{array}$$

onde y não ocorre livre em nenhum dos A_i e s deve identificar um objecto com um serviço op. Da interacção entre a mensagem op para s e o objecto localizado em s resulta uma mensagem da forma $y \lhd l_i : \tilde{v}_i$, com $1 \leq i \leq n$, que lançará o processo $A_i(\tilde{v}_i)$, onde os nomes em $\tilde{v}_i$ substituem os parâmetros de A_i.

Se s denota um valor[10] então pode-se escrever simplesmente:

$$\begin{array}{ll} \texttt{case}\ s\ \texttt{of} & \\ \quad l_1 : A_1 & \\ \quad \vdots & \\ \quad l_n : A_n & \\ \texttt{esac} & \stackrel{\text{def}}{=} \nu y\ s \lhd \mathsf{val} : y, y \rhd [l_1 : A_1\ \& \cdots \&\ l_n : A_n]) \end{array}$$

como já se tinha feito com a alternativa simples. ∎

Note-se que as alternativas simples que se definiram são casos particulares das generalizadas.

Listas e filas

Além dos naturais e dos booleanos serão úteis outros tipos de dados. Para expressar a semântica do TyCO em Gnome vão ser necessários conjuntos e filas, construidos à custa de células Nil e Cons. Estas estruturas são apresentadas a seguir.

Exemplo 3.1.10 *Construtores Nil e Cons*[11]
Os construtores Nil e Cons, aos quais se recorre usualmente para definir listas, podem ser implementados em TyCO pelos agentes seguintes:

$$\texttt{rec}\ \mathrm{Nil}\ = (l)\ l \rhd [\mathit{val} : (r)\ r \lhd \mathit{nil}, \ \mathrm{Nil}(l)]$$

[10] Ver, por exemplo, a definição feita atrás dos naturais
[11] Semelhantes aos definidos em [Vasconcelos 94d]

$$\texttt{rec}\ \mathrm{Cons}\ =\ (l\ h\ t)\ l \rhd [\mathit{val} : (r)\ r \lhd \mathit{cons} : h\ t,\ \mathrm{Cons}(l\ h\ t)]$$

O processo que representa a lista com dois elementos, a e b, localizada no nome l, é:

$$\nu s\ \mathrm{Cons}(l\ a\ s), \nu s'\ \mathrm{Cons}(s\ b\ s'), \mathrm{Nil}(s')$$

onde s e s' são nomes locais que servem apenas para "ligar" os elementos da lista. ■

A partir destes construtores podem-se definir estruturas mais complexas, como os conjuntos ou as filas.

Exemplo 3.1.11 *Conjuntos*
Na codificação das classes GNOME em TyCO, vão-se usar conjuntos para "guardar" os identificadores das instâncias existentes para uma dada classe.

Um "conjunto", com as operações *inserir*, *retirar* e *existe*, pode ser implementado na forma de lista, de acordo com o exemplo anterior, mas usando agora um "identificador escondido" ("dummy header") para simplificar as operações de inserção e anulação (de acordo com uma proposta de Vasconcelos).

$$
\begin{array}{l}
\texttt{rec}\ \mathrm{Conj}\ = (\mathit{self}\ d\ l) \\
\qquad \mathit{self} \rhd [\ \mathit{inserir} : (e\ r)\ \nu n\ \mathrm{Cons}(n\ e\ l), d \lhd \mathit{newtail} : n\ r, \mathrm{Conj}(\mathit{self}\ d\ n)\ \& \\
\qquad\qquad \mathit{existe} : (e\ r)\ \texttt{let rec}\ \mathrm{Ciclo}\ =\ (l) \\
\qquad\qquad\qquad \texttt{case}\ l\ \texttt{of} \\
\qquad\qquad\qquad\quad \mathit{nil} :\ r \lhd \mathit{false}, \mathrm{Conj}(\mathit{self}\ d\ l) \\
\qquad\qquad\qquad\quad \mathit{cons} : (h\ t)\ \texttt{if}\ \mathit{opnat} \lhd \mathit{eq} : e\ h\ \texttt{then} \\
\qquad\qquad\qquad\qquad\qquad r \lhd \mathit{true},\ \mathrm{Conj}(\mathit{self}\ d\ l) \\
\qquad\qquad\qquad\qquad \texttt{else} \\
\qquad\qquad\qquad\qquad\qquad \mathrm{Ciclo}(t) \\
\qquad\qquad\qquad\qquad \texttt{fi} \\
\qquad\qquad\qquad \texttt{esac} \\
\qquad\qquad\quad \texttt{in}\ \mathrm{Ciclo}(l)\ \& \\
\qquad\qquad \mathit{retirar} : (e\ r)\ \texttt{let rec}\ \mathrm{Ciclo}\ =\ (l_1\ l_2) \\
\qquad\qquad\qquad \texttt{case}\ l_2\ \texttt{of} \\
\qquad\qquad\qquad\quad \mathit{nil} :\ r \lhd \mathit{off},\ \mathrm{Conj}(\mathit{self}\ d\ l) \\
\qquad\qquad\qquad\quad \mathit{cons} : (h\ t)\ \texttt{if}\ \mathit{opnat} \lhd \mathit{eq} : e\ h\ \texttt{then} \\
\qquad\qquad\qquad\qquad\qquad l_2 \lhd \mathit{die}, l_1 \lhd \mathit{newtail} : t\ r, \\
\qquad\qquad\qquad\qquad\qquad \mathrm{Conj}(\mathit{self}\ d\ l) \\
\qquad\qquad\qquad\qquad \texttt{else} \\
\qquad\qquad\qquad\qquad\qquad \mathrm{Ciclo}(l_2\ t) \\
\qquad\qquad\qquad\qquad \texttt{fi} \\
\qquad\qquad\qquad \texttt{esac} \\
\qquad\qquad\quad \texttt{in}\ \mathrm{Ciclo}(d\ l)\ ]
\end{array}
$$

onde Cons é agora:

$$
\begin{array}{l}
\texttt{rec}\ \mathrm{Cons}\ =\ (l\ h\ t)\ l \rhd [\ \mathit{val} : (r)\ r \lhd \mathit{cons} : h\ t, \mathrm{Cons}(l\ h\ t)\ \& \\
\qquad\qquad\qquad\qquad\quad \mathit{newtail} : (t\ r)\ r \lhd \mathit{out}, \mathrm{Cons}(l\ h\ t)\ \& \\
\qquad\qquad\qquad\qquad\quad \mathit{die} : \mathbf{0}\ ]
\end{array}
$$

O serviço *inserir* coloca o novo elemento à cabeça da lista, através do serviço *inserir* da célula Cons que guarda a anterior cabeça da lista. É este segundo serviço que envia a mensagem $r \lhd out$ ao invocador de *inserir*, informando-o que a operação terminou (com sucesso) — este passo é necessário para implementar comunicação síncrona[12].

Quando se usar esta estrutura ver-se-à que não é necessário verificar se o elemento a inserir já existe no conjunto porque isso estará sempre garantido. Se não fosse o caso bastava começar por invocar o serviço *existe* da própria instância de Conj onde se pretende inserir.

O serviço *existe* consiste num processo iterativo que percorre a lista à procura do elemento em questão — se o encontra emite a mensagem $r \lhd true$; se não, emite a mensagem $r \lhd false$.

O serviço *retirar* consiste igualmente num processo iterativo que uma pesquisa sequencial do elemento em questão na lista, utilizando dois apontadores que marcam elementos consecutivos — se o apontador da frente encontra o elemento, a célula Cons que o contém é eliminada e "colam-se" as duas células que lhe eram adjacentes.

Supõe-se que está disponivel no sistema um objecto *opnat*, instância do agente que fornece as operações sobre os naturais; o seu serviço *eq* permite comparar nomes de processos, emitindo a mensagem *true* ou *false*, conforme esses nomes são iguais ou não.

A criação de uma instância de conjunto (vazio) faz-se recorrendo ao agente:

$$\text{Novo} \;=\; (\mathit{self})\ \nu d\ x\ n\ \text{Cons}(d\ x\ n), \text{Nil}(n), \text{Conj}(\mathit{self}\ d\ n)$$

■

Exemplo 3.1.12 *Filas*
Para implementar filas de sequências de nomes, com as operações *enqueue*, *dequeue* e *clean*, tem-se o seguinte agente:

$$
\begin{array}{l}
\texttt{rec}\ \text{Fila} \;=\; (\mathit{self}\ d\ f\ l) \\
\qquad\qquad \mathit{self} \rhd [\ \mathit{enqueue} : (\tilde{x}\ r)\ \nu n\ y\ \text{Cons}(n\ y\ \tilde{x}\ f), d \lhd \mathit{newtail} : n\ r, \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{Fila}(\mathit{self}\ d\ n\ l)\ \& \\
\qquad\qquad\qquad \mathit{dequeue} : (r)\ \texttt{case f of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{nil} :\ r \lhd \mathit{empty}, \text{Fila}(\mathit{self}\ d\ f\ l) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{cons} : (\tilde{x}\ n)\ f \lhd \mathit{die}, d \lhd \mathit{newtail} : n\ r, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad r \lhd \mathit{dequeued} : \tilde{x}, \text{Fila}(\mathit{self}\ d\ f\ n) \\
\qquad\qquad\qquad\qquad\qquad \texttt{esac}\ \& \\
\qquad\qquad\qquad \mathit{clean} : \texttt{let rec}\ \text{Ciclo} \;=\; (l_1\ l_2) \\
\qquad\qquad\qquad\qquad\qquad \texttt{case}\ l_2 \lhd \mathit{dequeue}\ \texttt{of} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{dequeued} : (l\ p\ r)\ l_2 \lhd \mathit{die}, l_1 \lhd \mathit{newtail} : t\ r, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{Ciclo}(l_2\ l) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{empty} : \mathbf{0} \\
\qquad\qquad\qquad\qquad\qquad \texttt{esac} \\
\qquad\qquad\qquad\qquad \texttt{in}\ \text{Ciclo}(d\ f)\ ]
\end{array}
$$

[12] Ver definição 3.1.12 e sua aplicação no capítulo 4

onde Cons é agora

$$\begin{aligned}\texttt{rec}\ \mathrm{Cons} \ = \ (f\ p\ \tilde{e}\ n)\ f \triangleright [\,& \mathit{val} : (r)\ r \triangleleft \mathit{cons} : p\ \tilde{e}\ n,\ \mathrm{Cons}(f\ p\ \tilde{e}\ n)\ \& \\ & \mathit{newtail} : (t\ r)\ r \triangleleft \mathit{out}, \mathrm{Cons}(f\ p\ \tilde{e}\ t)\ \& \\ & \mathit{die} : \mathbf{0}\,]\end{aligned}$$

A fila é duplamente encadeada – tem um apontador para a primeira posição e outro para a última, e cada célula Cons tem um apontador para a frente e outro para trás — e tem também um identificador escondido. Por isso, os seus serviços de inserção e anulação são semelhantes aos do conjunto, com a diferença que, quando se retira não é necessário saber o elemento porque será sempre o que está na última posição[13].

A criação de instâncias destas filas (vazias) é novamente feita por instanciação do termo de agente seguinte com um nome de processo:

$$\mathrm{Novafila} \ = \ (\mathit{self})\ \nu n\ d\ x\ \mathrm{Nil}(n), \mathrm{Cons}(d\ n\ x\ n),\ \mathrm{Fila}(\mathit{self}\ d\ n\ n)$$

■

Sincronização de processos

Definição 3.1.13 Dois processos, P e Q, são executados sequencialmente e sem possibilidades de interferência exterior na sua execução, do modo seguinte:

$$P; Q \stackrel{\mathrm{def}}{=} \nu c\ P, c \triangleright [\ \mathit{out} : (\tilde{v})\ Q]$$

onde P eventualmente emitirá a mensagem $c \triangleleft \mathit{out} : \tilde{a}$, sendo $\tilde{a}$ a sequência de nomes de processos que contém os resultados da avaliação de P e c o nome que sincroniza P e Q. Se P não enviar nenhuma mensagem ao nome c, o processo Q não será lançado.

Pode-se usar uma definição mais completa, para os casos em que se quer sincronizar o processo P com dois processos (Q e R) em alternativa. Agora P tem que emitir, ou a mensagem $c \triangleleft \mathit{out} : \tilde{v}$, que lança o processo Q, ou a mensagem $c \triangleleft \mathit{off} : \tilde{u}$, que lança o processo R:

$$P; (Q \mid R) \stackrel{\mathrm{def}}{=} \nu c\ P, c \triangleright [\ \mathit{out} : (\tilde{x})\ Q\ \&\ \mathit{off} : (\tilde{y})\ R]$$

Convenciona-se que c, d e e são nomes usados para sincronização de processos, v, u, são nomes que conterão valores resultantes da avaliação de expressões e x e y serão parâmetros de processos. ■

3.2 O cálculo de objectos

Define-se uma semântica operacional para o TyCO à custa de duas relações: a congruência estrutural e a redução.

[13] A disciplina de acesso aos elementos da fila é FIFO: First In First Out

3.2.1 Congruência e redução

Definição 3.2.1 A ocorrência de um nome de processo x num processo P diz-se *muda* se:

- P ocorre no método de um serviço de um objecto, tal como $l\!:\!(\tilde{w}x\tilde{y})P$, com $l \in \mathbf{L}$ e $\tilde{w}, \tilde{y} \in \mathbf{NP}^*$;
- P ocorre no âmbito de restrição de x, num processo $\nu x P$.

Caso contrário a ocorrência de x em P diz-se *livre*. O conjunto dos nomes de processos livres no processo P ou no agente A é denotado por $\mathbf{NL}(P)$ ou $\mathbf{NL}(A)$.

Da mesma forma se define a ocorrência muda de uma nome de agente X:

- num agente A, se ocorre num termo da forma `rec` $X.A$;
- num processo P, se ocorre num termo da forma `let` $X = A$ `in` P, sendo A um agente.

Igualmente se define o conjunto dos nomes de agentes livres num processo ou num agente, denotado por $\mathbf{AL}(P)$ ou $\mathbf{AL}(A)$. ∎

A recepção de uma mensagem por um objecto constitui o mecanismo básico de *computação* no cálculo — a **comunicação**. Esta interacção entre uma mensagem e um objecto, que partilham um nome de processo, desencadeia a execução dum método. A operação chama-se **aplicação de mensagem** e é definida como se segue:

Definição 3.2.2 Seja C a comunicação $l_i : \tilde{v}$ de determinada mensagem para um nome de processo e seja M a colecção de serviços $[l_1 : A_1 \;\&\cdots\&\; l_n : A_n]$ de um objecto com o mesmo nome.

A *aplicação* da mensagem C à colecção de métodos M, denotada por $M \bullet C$, é uma operação que define um processo onde a ocorrência do agente $(\tilde{x}_i)P_i$ é substituida pelo processo $P_i\{\tilde{v}/\tilde{x}_i\}$, desde que $1 \leq i \leq n$ e que o comprimento da sequência $\tilde{v}$ seja igual ao comprimento da sequência $\tilde{x}_i$, para que a substituição dos parâmetros do termo de agente pelos nomes de processos seja possível. ∎

A etiqueta l_i da mensagem selecciona um serviço no objecto e no método A_i desse serviço, os nomes em $\tilde{v}$ substituem as ocorrências livres dos nomes de processos em $\tilde{x}_i$ (parâmetros de um processo P_i), sendo depois lançado o processo $P_i\{\tilde{v}/\tilde{x}_i\}$.

A substituição das ocorrências livres do nome de agente X pelo agente A no processo P denota-se por $P[X := A]$.

Definição 3.2.3 Sejam P e Q termos quaisquer; o conjunto das *fórmulas* é definido indutivamente do modo que se segue:

- $P \equiv Q$, que é a relação de **congruência estrutural**. É a menor relação de congruência sobre P que respeita as regras que se apresentam a seguir;

- $P \rightarrow Q$, que é a relação de **redução**. Significa que P se reduz a Q num só passo computacional. ∎

Definição 3.2.4 Dois termos quaisquer, P e Q, que diferem apenas nos nomes (de processos ou agentes) mudos, dizem-se em *congruência alfa* ($P \equiv_\alpha Q$) [Barendregt 74].

O conjunto **CO**, das *fórmulas derivadas*, é definido indutivamente do modo que se segue:

1. $P \equiv Q \in \mathbf{CO}$ desde que $P \equiv_\alpha Q$
2. $P, Q \equiv Q, P \in \mathbf{CO}$ e $P, (Q, R) \equiv (P, Q), R \in \mathbf{CO}$
3. $a \rhd M_1 \equiv a \rhd M_2 \in \mathbf{CO}$ desde que M_2 seja uma permutação de M_1
4. $\nu x P, Q \equiv \nu x(P, Q) \in \mathbf{CO}$ e $\nu x Q \equiv Q \in CO$ desde que $x \notin \mathbf{NL}(Q)$
5. $((\tilde{x})P)(\tilde{v}) \equiv P\{\tilde{v}/\tilde{x}\} \in \mathbf{CO}$
6. $\texttt{rec}\ X.A \equiv A[X := \texttt{rec}\ X.A] \in \mathbf{CO}$
7. $\texttt{let}\ X = A\ \texttt{in}\ P \equiv P[X := A] \in \mathbf{CO}$
8. $\nu \tilde{x}(a \rhd M, a \lhd C, \tilde{P}) \rightarrow \nu \tilde{x}(M \bullet C, \tilde{P})$
9. $P' \rightarrow Q' \in \mathbf{CO}$ desde que $P' \equiv P, P \rightarrow Q, Q \equiv Q' \in \mathbf{CO}$ ∎

As 7 primeiras regras definem a relação de *congruência estrutural* ($\equiv$). As duas últimas regras definem a relação de *redução simples* ($\rightarrow$). A *redução* ($\twoheadrightarrow$) é a relação ($\equiv \cup \rightarrow^*$), onde "$\rightarrow^*$" denota o fecho trânsitivo de "$\rightarrow$".

3.2.2 Exemplos

Definido o cálculo ilustra-se o mecanismo de computação de processos para alguns dos exemplos já apresentados.

Exemplo 3.2.5 *Avaliação de uma alternativa simples*
Considere-se que estão em paralelo um objecto (persistente), instância do agente True[14], e o processo que representa a alternativa simples[15]. Observe-se a interacção entre eles:

$$
\begin{array}{ll}
\mathrm{True}(b), \texttt{if}\ b\ \texttt{then}\ P\ \texttt{else}\ Q\ \texttt{fi} \stackrel{\mathrm{def}}{=} & \\
b \rhd [\mathit{val} : (r)\ r \lhd \mathit{true}], \nu x\ b \lhd \mathit{val} : x, x \rhd [\mathit{true} : P\ \&\ \mathit{false} : Q] \equiv & \text{(regra 4)} \\
\nu x\ b \rhd [\mathit{val} : (r)\ r \lhd \mathit{true}], b \lhd \mathit{val} : x, x \rhd [\mathit{true} : P\ \&\ \mathit{false} : Q] \rightarrow & \text{(regra 8)} \\
\nu x\ x \lhd \mathit{true}, x \rhd [\mathit{true} : P\ \&\ \mathit{false} : Q] \rightarrow & \text{(regra 2)} \\
\nu x\ x \rhd [\mathit{true} : P\ \&\ \mathit{false} : Q], x \lhd \mathit{true} \rightarrow & \text{(regra 8)} \\
\nu x\ P \equiv & \text{(regra 4)} \\
P &
\end{array}
$$

[14] Ver exemplo 3.1.5
[15] Ver definição 3.1.8

Se o objecto localizado em b for instância do agente False resulta agora:

$$\begin{array}{l}\mathrm{False}(b), \texttt{if}\ b\ \texttt{then}\ P\ \texttt{else}\ Q\ \texttt{fi}\ \stackrel{\mathrm{def}}{=} \\ b \rhd [\mathit{val} : (r)\ r \lhd \mathit{false}], \nu x\ b \lhd \mathit{val} : x, x \rhd [\mathit{true} : P\ \&\ \mathit{false} : Q] \twoheadrightarrow Q\end{array}$$

■

Exemplo 3.2.6 *Operações sobre os naturais*
Pretende-se calcular o predecessor de um dado natural (por exemplo, 1). Supondo-se que existem no sistema objectos que representam os naturais[16] e um objecto que fornece operações sobre os naturais, o cálculo faz-se como se segue, aplicando as regras 8 e 9:

$$\begin{array}{l}\mathrm{Zero}(z), \mathrm{Suc}(1\ z), o \lhd \mathit{pred} : 1\ c, \mathrm{Natop}(o)\ \stackrel{\mathrm{def}}{=} \\ \mathrm{Zero}(z), 1 \rhd [\mathit{val} : (r)\ r \lhd \mathit{suc} : z, \mathrm{Suc}(1\ z)], o \lhd \mathit{pred} : 1\ c, \\ o \rhd [\mathit{pred} : (n\ r)\ \nu x\ n \lhd \mathit{val} : x, x \rhd [\mathit{suc} : (v)\ r \lhd \mathit{val} : v, \mathrm{Natop}(o)\ ]\ \&\ \cdots] \rightarrow \\ \mathrm{Zero}(z), 1 \rhd [\mathit{val} : (r)\ r \lhd \mathit{suc} : z, \mathrm{Suc}(1\ z)], \nu x\ 1 \lhd \mathit{val} : x, \\ x \rhd [\mathit{suc} : (v)\ r \lhd \mathit{val} : v, \mathrm{Natop}(o)\ ]\ \&\ \cdots] \rightarrow \\ \mathrm{Zero}(z), x \lhd \mathit{suc} : z, \mathrm{Suc}(1\ z), x \rhd [\mathit{suc} : (v)\ r \lhd \mathit{val} : v, \mathrm{Natop}(o)\ ]\ \&\ \cdots] \rightarrow \\ \mathrm{Zero}(z), \mathrm{Suc}(11\ zz), c \lhd \mathit{val} : z, \mathrm{Natop}(o)\end{array}$$

Resulta uma mensagem para o invocador da operação, que transporta o resultado. ■

Exemplo 3.2.7 *Inserção e anulação num conjunto*
Considere-se um conjunto vazio[17] e suponha-se que se pretende inserir um elemento nesse conjunto.

$$\begin{array}{l}\nu d\ x\ l\ \mathrm{Cons}(d\ x\ l), \mathrm{Nil}(l), \mathrm{Conj}(c\ d\ l), c \lhd \mathit{inserir} : u\ a \rightarrow \\ \nu d\ x\ l\ n\ \mathrm{Cons}(n\ u\ l), d \lhd \mathit{newtail} : n\ a, \mathrm{Cons}(d\ x\ l), \mathrm{Nil}(l), \mathrm{Conj}(c\ d\ n) \rightarrow \\ \nu d\ x\ l\ n\ \mathrm{Cons}(n\ u\ l), \mathrm{Cons}(d\ x\ n), \mathrm{Nil}(l), \mathrm{Conj}(c\ d\ n), a \lhd \mathit{out}\end{array}$$

A mensagem $a \lhd \mathit{out}$ indica ao invocador que o serviço foi executado com sucesso.

Se se quiser anular um elemento do conjunto, tem-se:

$$\begin{array}{l}\nu d\ x\ l\ n\ m\ \mathrm{Cons}(d\ x\ m), \mathrm{Cons}(m\ v\ n), \mathrm{Cons}(n\ u\ l), \mathrm{Nil}(l), \mathrm{Conj}(c\ d\ m) \rightarrow \\ \mathrm{Natop}(o), c \lhd \mathit{retirar} : u\ a \\ \mathrm{Natop}(o), \nu d\ x\ l\ n\ m\ \mathrm{Cons}(d\ x\ m), \mathrm{Cons}(m\ v\ n), \mathrm{Cons}(n\ u\ l), \mathrm{Nil}(l), \mathrm{Ciclo}(d\ m) \rightarrow \\ \mathrm{Natop}(o), \nu d\ x\ l\ n\ m\ \mathrm{Cons}(d\ x\ m), \mathrm{Cons}(m\ v\ n), \mathrm{Cons}(n\ u\ l), \mathrm{Nil}(l), \\ \texttt{if}\ o \lhd \mathit{eq} : u\ v\ \texttt{then}\ m \lhd \mathit{die}, d \lhd \mathit{newtail} : n\ a, \mathrm{Conj}(c\ d\ m)\ \texttt{else}\ \mathrm{Ciclo}(m\ n)\ \texttt{fi} \rightarrow \\ \mathrm{Natop}(o), \nu d\ x\ l\ n\ m\ \mathrm{Cons}(d\ x\ m), \mathrm{Cons}(m\ v\ n), \mathrm{Cons}(n\ u\ l), \mathrm{Nil}(l), \mathrm{Ciclo}(m\ n) \rightarrow \\ \mathrm{Natop}(o), \nu d\ x\ l\ n\ m\ \mathrm{Cons}(d\ x\ m), \mathrm{Cons}(m\ v\ n), \mathrm{Cons}(n\ u\ l), \mathrm{Nil}(l), \\ \texttt{if}\ o \lhd \mathit{eq} : u\ u\ \texttt{then}\ n \lhd \mathit{die}, d \lhd \mathit{newtail} : l\ a, \mathrm{Conj}(c\ d\ m)\ \texttt{else}\ \mathrm{Ciclo}(n\ l)\ \texttt{fi} \rightarrow \\ \mathrm{Natop}(o), \nu d\ x\ l\ n\ m\ \mathrm{Cons}(d\ x\ m), \mathrm{Cons}(m\ v\ n), \mathrm{Cons}(n\ u\ l), \mathrm{Nil}(l), \\ n \lhd \mathit{die}, d \lhd \mathit{newtail} : n\ a, \mathrm{Conj}(c\ d\ m) \twoheadrightarrow \\ \mathrm{Natop}(o), \nu d\ x\ l\ n\ m\ \mathrm{Cons}(d\ x\ m), \mathrm{Cons}(m\ v\ l), \mathrm{Nil}(l), \mathrm{Conj}(c\ d\ m), a \lhd \mathit{out}\end{array}$$

■

[16] De acordo com a definição 3.1.6
[17] Ver a proposta feita no final do exemplo 3.1.11

Capítulo 4

Semântica da linguagem GNOME em TyCO

4.1 A metodologia de codificação

4.1.1 Objectivo

Um dos objectivos desta dissertação é expressar a semântica operacional do GNOME em TyCO. Para tal, uma vez que o GNOME é baseado em classes, é necessário codificar as classes em termos TyCO.

Uma fonte de inspiração foi [Vasconcelos 94a], onde igualmente se codifica em TyCO uma linguagem de programação concorrente e orientada a objectos (o ABCL/1 [yon90]). Nesse trabalho a codificação foi feita primitiva a primitiva. Aqui a estratégia terá de ser diferente, pois o GNOME é uma linguagem mais elaborada. Optou-se por partir de uma especificação GNOME, que define uma comunidade de objectos, e construir um agente TyCO de forma que as suas instâncias tenham um comportamento equivalente a essa comunidade de objectos.

Basta ver como representar as classes, pois uma especificação, no fragmento de GNOME considerado, é simplesmente um conjunto de classes. Assim, cada classe é codificada naturalmente num agente, sendo os objectos, instâncias da classe, representados por processos, instâncias do agente.

Mas representar uma classe GNOME em TyCO é uma tarefa complexa, que implica muito mais que traduzir os métodos das suas diversas acções (internas ou externas) cláusula a cláusula, e compôr (em concorrência) os termos obtidos em agentes que representem as classes. Para ser possível afirmar que as instâncias de determinado agente tem um comportamento equivalente à comunidade de objectos que a especificação define, é necessário construir processos que emulem a semântica dos objectos em GNOME. Esses processos serão instâncias de agentes TyCO que representam as classes GNOME, sendo possíveis, em ambos, as mesmas transacções.

4.1.2 O mapa de tradução

Pretende-se descrever genericamente o mapa de tradução que permitirá expressar a semântica do GNOME em TyCO. O mapa de tradução é denotado por $[\![\cdot]\!]$. Estabelece-se que as classes GNOME, por serem moldes para a criação de objectos, serão representadas por agentes TyCO (famílias de processos parametrizados por nomes de processo), e naturalmente, os objectos serão representados por instâncias desses agentes.

Importa ver ainda como o mapa de tradução faz a correspondência entre os elementos das linguagens e a codificação das primitivas. Apresentam-se ambas seguidamente.

Pressupõe-se que o código GNOME a traduzir está correcto do ponto de vista sintáctico e está também com uma tipificação coerente; considera-se disponível uma assinatura de tipos de dados que contemple os tipos do GNOME, já devidamente codificados em TyCO[1].

Correspondência entre as linguagens

Para fazer a correspondência entre os elementos da linguagem do TyCO e as principais entidades do GNOME agrupam-se estas últimas em três tipos: as que identificam memória (os identificadores de atributos, parâmetros e resultados), as que identificam acções (internas ou externas) e as que identificam operações da assinatura de tipos de dados. O conjunto **NP** dos nomes de processo vai conter os identificadores dos atributos, os nomes das instâncias das classes, os nomes das instâncias dos agentes que intervêm no agente que representa a classe e outros nomes auxiliares; o conjunto **NA** dos nomes de agentes vai conter os identificadores dos agentes que sejam componentes do agente que representa a classe e o identificador deste último agente, e os identificadores das operações sobre os tipos de dados disponíveis; os identificadores das acções (internas ou externas) e os valores e operações dos diversos tipos de dados serão etiquetas, elementos de **L**.

Codificação das primitivas

O segundo aspecto da tradução é a codificação das classes: o agente TyCO que representa uma classe GNOME será denotado por $[\![\text{Classe}]\!]$. Para obter esse agente é preciso codificar as diversas primitivas que podem intervir numa classe, nomeadamente, as cláusulas dos seus métodos. Mas como se verá, não é possível traduzir uma cláusula independentemente do contexto onde surge. A tradução só terá sentido se for de todo o método, estando este inserido na respectiva classe. Vai ser também necessário codificar as expressões que surgem em algumas dessas cláusulas.

Na próxima secção explica-se a abordagem escolhida para apresentar a tradução das classes. Nas secções seguintes fazem-se sucessivas codificações de classes, de forma a codificar todas a primitivas.

4.1.3 Representação de uma classe

Uma vez que o agente que representa uma classe é complexo, não sendo fácil, nem explicar, de uma vez só, todo o seu comportamento, nem justificar, conjuntamente, todas as opções tomadas na sua implementação, optou-se por seguir uma metodologia "construtiva" na sua apresentação: as diversas primitivas do GNOME serão introduzidas gradualmente, inseridas em diferentes classes; começa-se por fazer codificações "simplistas" dessas classes, criticam-se depois os agentes obtidos e progressivamente vão-se refinando esses agentes,

[1] Ver secção 3.1.3

até se chegar a um agente final que represente uma classe com estrutura típica (que envolva todas as primitivas do fragmento da linguagem GNOME que se pretende tratar).

Padrão da apresentação

As sucessivas apresentações dos agentes que representam as classes propostas para introduzir cada primitiva vão seguir um padrão: propõe-se primeiro um exemplo de uma classe e o respectivo agente que a representa, em seguida discute-se o seu comportamento (operacionalmente), e finalmente, generaliza-se o agente para uma classe com estrutura típica (considerando as primitivas que se estão a tratar).

Estrutura típica de uma classe

Uma classe compreende um conjunto de acções, cada uma com um ou mais métodos, em alternativa. Um método é um conjunto de cláusulas, declaradas por uma ordem determinada: primeiro vêm as condições de activação, depois as invocações de serviços (de outras instâncias), depois as valorações dos atributos e, finalmente, os retornos de valores (retornos estes que não estão disponíveis para as acções internas).

Para simplificar o mapa de tradução introduzem-se restrições à estrutura típica de uma classe GNOME[2], sem haver com isso perda de generalidade:

- considera-se apenas um atributo por classe e, no máximo, um parâmetro e um resultado por serviço;
- considera-se que cada método não terá mais que uma condição de activação (sendo eventualmente a conjunção de várias condições). Pode no entanto ter várias chamadas e, atendendo à restrição anterior, terá, no máximo, uma só valoração e um só retorno de valores;
- não se consideram alternativas nas acções internas[3].

Semântica da execução de um método

A tradução de um método é a tradução de um conjunto de cláusulas. Dois pressupostos sobre a execução de um método tem que ser garantidos para se obedecer à semântica da linguagem:

- um método só está activado se forem verdadeiras todas as suas cláusulas de activação e se estiverem activadas as acções chamadas no seu corpo;
- a sua execução deve ser atómica, pressupondo-se que a avaliação das diversas cláusulas que o compõem não pode ser interrompida e que não pode haver mudanças de estado durante essa execução.

[2] A especificação de uma classe com estrutura típica é apresentada no final do Apêndice A.

[3] Para explicar o tratamento das alternativa basta considerá-las nos serviços, onde são de facto indispensáveis

Logo, deve-se fazer, em primeiro lugar, a verificação da activação do método: avalia-se a condição de activação e, se esta for verdadeira, verifica-se se os serviços chamados estão activados; se for o caso, executam-se em paralelo esses serviços, passando um valor para cada parâmetro de cada serviço, e recebendo depois os resultados de cada um. Em segundo lugar, executam-se, em paralelo, a valoração do atributo e o retorno do resultado.

Para que a execução de um método seja atómica as suas cláusulas devem ser executadas sem permitir intercalações.

4.2 Classes de objectos persistentes e deterministas

A partir de agora, o capítulo seguirá o padrão proposto atrás, até que se obtenha um agente que represente uma classe com estrutura típica.

Numa primeira aproximação, os objectos serão persistentes e deterministas.

4.2.1 Serviços com valoração e retorno

Começa-se por considerar classes só com serviços, sem serem nem de nascimento nem de morte, e os métodos destes são simples e não incluem cláusulas de chamada.

O agente TyCO que vai representar a classe GNOME nas condições estabelecidas deve disponibilizar os serviços dessa classe. Imediatamente, a opção de codificação mais simples parece ser a de definir um agente que tenha como parâmetros a identidade da instância e o atributo, e cujo processo seja um objecto cujos métodos correspondam aos serviços da classe.

Exemplo 4.2.1 *Contador*
Considere-se a seguinte classe GNOME, que implementa um "contador" de naturais:

⟦`class` Cont
  `interface`
    `serv` *inc*
    `serv` *set*
      `par` *p*: `nat`
    `serv` *obs*
      `res` *r*: `nat`
  `body`
    `slot` *n*: `nat`
    `serv` *inc*
      `val` $n \ll n+1$
    `serv` *set*
      `val` $n \ll set.p$
    `serv` *obs*
      `ret` $obs.r$ `=` n
`end` Cont⟧ =

$$\texttt{rec}\ \mathrm{Cont} = (self\, n)\ self \rhd [\, inc : \mathrm{Cont}(self\, [\![n+1]\!]) \ \&$$
$$set : (p)\ \mathrm{Cont}(self\, p)\ \&$$
$$obs : (c)\ c \lhd out : n, \mathrm{Cont}(self\, n)]$$

onde se tem:

$\text{Cont}(\mathit{self}\,[\![n+1]\!]) \stackrel{\text{def}}{=} \nu n'\ [\![n+1]\!]_{n'}\text{Cont}(\mathit{self}\, n') \stackrel{\text{def}}{=}$
$\nu x\ op \lhd \mathit{add} : n\ 1\ x, x \rhd [\mathit{val} : (n')\text{Cont}(\mathit{self}\, n')]$ ∎

Como se depreende do exemplo, um pedido de serviço é uma mensagem que tem como alvo a instância invocada e que selecciona o método que corresponde ao serviço desejado (no objecto que representa a instância). Se é pedido o serviço *inc*, a mensagem lança o processo que actualiza o valor do atributo, supondo que existe no ambiente, em paralelo com a instância, um objecto com nome *op*, que tem como métodos as operações sobre os naturais; se é pedido o serviço *set*, a mensagem transporta um nome (que denota um valor) para instanciar o parâmetro do serviço; se é pedido o serviço *obs*, a mensagem transporta um nome para onde deve ser enviada a resposta que consiste numa mensagem que transporta o nome que denota o atributo.

Analise-se em maior detalhe as implicações da codificação de cada serviço: se i for um nome que denota uma instância da classe Cont, que tem guardado o valor 3 (no atributo n), e Natop o agente proposto em 3.1.6, o seu comportamento face aos pedidos de execução dos serviços será:

inc: A execução deste serviço implica avaliar a expressão de valoração e actualizar o atributo com o resultado dessa avaliação.

$\text{Cont}(i\ n), i \lhd \mathit{inc}, \text{Natop}(o) \stackrel{\text{def}}{=}$
$\nu n'\ [\![n+1]\!]_{n'}\ \text{Cont}(i\ n'), i \lhd \mathit{inc}, \text{Natop}(o) \stackrel{\text{def}}{=}$
$\nu x\ o \lhd \mathit{add} : n\ [\![1]\!]\ x, x \rhd [\mathit{val} : (n')\text{Cont}(i\ n')], \text{Natop}(o) \rightarrow$
$\nu x\ x \lhd \mathit{val} : [\![4]\!], x \rhd [\mathit{val} : (n')\text{Cont}(i\ n')], \text{Natop}(o) \rightarrow$
$\text{Cont}(i\ [\![4]\!]), \text{Natop}(o)$

onde $[\![1]\!]$ e $[\![4]\!]$ denotam nomes de processo que representam os valores naturais 1 e 4.

set: Na execução deste serviço, o valor passado no seu parâmetro instancia o atributo.

$\text{Cont}(i\ n), i \lhd \mathit{set} : [\![0]\!] \rightarrow\ \text{Cont}(i\ [\![0]\!])$

obs: Ao invocar-se este serviço pretende-se saber o valor do atributo: a mensagem que desencadeia a sua execução passa um nome que será o alvo da mensagem que transporta esse valor.

$\text{Cont}(i\ n), i \lhd \mathit{obs} : j \rightarrow\ j \lhd \mathit{out} : n,\ \text{Cont}(i\ n)$

O passo seguinte é generalizar esta codificação das cláusulas de valoração e de retorno de valores, juntá-las em métodos com condições de activação e ver que aspecto assumem os agentes que representam as classes com tais métodos. Necessáriamente, tem que se ver como traduzir as expressões.

Avaliação de expressões

De acordo com a implementação dos tipos de dados apresentada na secção 3.1.3, podem-se considerar dois tipos de expressões: as constantes (operações de aridade zero) e as que envolvem operações de aridade superior a zero. Veja-se separadamente como se faz a avaliação de cada uma delas.

constantes seja c uma constante de determinado tipo de dados; a avaliação da constante c (denotada por $[\![c]\!]_u$) é o processo TyCO:

$$[\![\mathrm{c}]\!]_u \stackrel{\mathrm{def}}{=} \nu u\ c \lhd \mathit{val} : u$$

onde c é um nome (de processo) que identifica um objecto que representa essa constante e u outro nome para onde deve ser enviada a mensagem com o "valor" da constante. Supõe-se que existe, no ambiente, o referido objecto, com nome c[4].

expressões considere-se que envolvem apenas uma operação, op, com argumentos $\tilde{x}$; a avaliação da expressão (denotada por $[\![\mathrm{op}(\tilde{x})]\!]_c$) é o processo TyCO:

$$[\![\mathrm{op}(\tilde{x})]\!]_c \stackrel{\mathrm{def}}{=} \nu y\ op \lhd op : \tilde{x}y, y \rhd [\mathit{val} : (c)P]$$

onde op denota um objecto que disponibiliza as operações do tipo de dados da expressão[5] (que se pressupõe em paralelo com o processo definido acima), e P representa a restante codificação da classe onde a cláusula ocorre.

Valoração de atributos

Trata-se de um construtor que funciona como um comando de atribuição — instancia o atributo com o valor obtido pela avaliação da expressão.

Uma vez que a execução de um método deve ser atómica e sem mudanças intermédias de estado, durante a execução o atributo terá, no máximo, uma mudança de estado. Como uma classe é representada por um agente C, em que o primeiro parâmetro é a identidade da instância e o segundo é o atributo, a valoração implica a alteração do valor desse atributo, que deve receber o nome que identifica um objecto (que representa o valor resultante da avaliação da expressão de valoração[6]):

$$[\![\texttt{val}\ \mathit{id\_slot}\ \ll \mathbf{express\tilde{a}o}]\!]_c = \mathrm{C}(\mathit{self}\,[\![\mathbf{express\tilde{a}o}]\!]) \stackrel{\mathrm{def}}{=} \nu u\ [\![\mathbf{express\tilde{a}o}]\!]_u\ \mathrm{C}(\mathit{self}\,u)$$

De acordo com a definição da codificação das expressões, $[\![\mathbf{express\tilde{a}o}]\!]$ denota um nome que representa o valor resultante da avaliação da expressão e $[\![\mathbf{express\tilde{a}o}]\!]_u$ denota o processo que avalia a expressão e envia o resultado ao nome de continuação u.

A instância da classe é representada pelo processo $\mathrm{C}(\mathit{self}\ \mathit{id\_slot})$, onde *self* é a identificação dessa instância e *id_slot* o seu atributo.

Retorno de valores

Através deste construtor faz-se o retorno de valores à instância que invocou o serviço (só está disponível para acções externas pois faz parte do mecanismo de troca de mensagens).

[4] Tem estrutura semelhante a alguns objectos apresentados no Capítulo 3 (secção 3.1.3).

[5] De acordo com o que é feito na secção 3.1.3

[6] A partir de agora, quando se falar de valores refere-se os nomes que denotam objectos que representam esses valores.

O valor resultante da avaliação da expressão é enviado ao nome de continuação, c, que foi passado pelo invocador do serviço, recebendo assim este o resultado:

$$[\![\texttt{ret}\ id\_res\ = \textbf{expressão}]\!]_c = c \lhd out : [\![\textbf{expressão}]\!] \stackrel{\text{def}}{=} \nu u\ [\![\textbf{expressão}]\!]_u (c \lhd out : u)$$

A mensagem *out* transporta o nome que denota o resultado da avaliação da expressão.

Condições de activação

Esta cláusula é uma alternativa simples[7]: se é verdadeira executam-se as restantes cláusulas (valoração e retorno) do método onde ocorre; se é falsa, o método onde ocorre a cláusula não está activado.

$$\begin{array}{ll} [\![\texttt{enb}\ \textbf{condição} & \\ \quad \texttt{val}\ \textbf{exp\_val} & \\ \quad \texttt{ret}\ \textbf{exp\_ret}]\!]_c = & \texttt{if}\ [\![\textbf{condição}]\!]\ \texttt{then} \\ & \quad c \lhd out : [\![\textbf{exp\_ret}]\!], \mathrm{Class}(self\,[\![\textbf{exp\_val}]\!])) \\ & \texttt{else} \\ & \quad c \lhd off, \mathrm{Class}(self\ id\_slot) \\ & \texttt{fi} \end{array}$$

Quando se executa um método avalia-se primeiro a sua condição de activação: se é verdadeira (a acção está activada), são lançadas, em paralelo, as restantes cláusulas (valoração do atributo e retorno do resultado), que são codificadas como se viu atrás; se é falsa (a acção não está activada), não se permite que a execução do método continue: envia-se ao nome de continuação (c, passado pelo invocador) a mensagem *off*, que indica que o serviço não está activado, e repõe-se o processo que representa a instância, sem se alterar o atributo.

A classe

Apresenta-se agora o agente que representa uma classe com a configuração definida no início da secção, de acordo com o mapa de tradução definido:

$$\begin{array}{l} [\![\texttt{class}\ \mathrm{Class} \\ \quad \texttt{interface} \\ \quad\quad \texttt{serv}\ id\_serv \\ \quad\quad\quad \texttt{par}\ id\_par\text{:}\ id\_tipo \\ \quad\quad\quad \texttt{res}\ id\_res\text{:}\ id\_tipo \\ \quad \texttt{body} \\ \quad\quad \texttt{slot}\ id\_slot\text{:}\ id\_tipo \\ \quad\quad \texttt{serv}\ id\_serv \\ \quad\quad\quad \texttt{enb}\ \textbf{condição} \\ \quad\quad\quad \texttt{val}\ id\_slot \ll \textbf{exp\_val} \\ \quad\quad\quad \texttt{ret}\ id\_res\ \texttt{=}\ \textbf{exp\_ret} \\ \texttt{end}\ \mathrm{Class}]\!] = \end{array}$$

[7] Usa-se a proposta da definição 3.1.8.

$$
\begin{array}{l}
\texttt{rec}\ \mathrm{Class} = (\mathit{self}\ \mathit{id\_slot}) \\
\qquad \mathit{self} \rhd [\mathit{id\_serv} : (\mathit{id\_par}\ c)\,\texttt{if}\ [\![\textbf{condição}]\!]\ \texttt{then} \\
\qquad\qquad\qquad\qquad\qquad\quad c \lhd \mathit{out} : [\![\textbf{exp\_ret}]\!], \mathrm{Class}(\mathit{self}\,[\![\textbf{exp\_val}]\!])) \\
\qquad\qquad\qquad\qquad\qquad\ \texttt{else} \\
\qquad\qquad\qquad\qquad\qquad\quad c \lhd \mathit{off}, \mathrm{Class}(\mathit{self}\ \mathit{id\_slot}) \\
\qquad\qquad\qquad\qquad\qquad\ \texttt{fi}]
\end{array}
$$

Se a classe tivesse outros serviços, eles seriam métodos do objecto *self* do agente Class. Repare-se que o nome *id_res* não é usado no agente (é desnecessário para o retorno de valores em TyCO).

4.2.2 Serviços com chamada

Considere-se que os serviços podem ter invocações de outros serviços (de outras instâncias). Os métodos passam a ter a estrutura completa que apresentam em Gnome.

Semântica da invocação de serviços

A implementação anterior da atomicidade da execução de um método pode trazer problemas: se a instância que chama um destes serviços o faz também através de um serviço, que não está activado (chama, por exemplo, outro serviço, que não está activado), deve-se desistir da execução desse serviço chamado. Assim não é possivel, pois uma vez pedida a execução não se pode voltar atrás, estando eventualmente o atributo já alterado e o resultado enviado.

Para se obter o comportamento descrito é necessário, antes de executar o serviço, verificar da sua activação, comunicar o seu estado ao invocador e, no caso de estar activado, esperar uma confirmação para executar (ou desistir dessa execução).

Sistematizando, a chamada de um serviço deve ser feita em dois tempos:

- primeiro verifica-se da sua activação: o serviço invocado responde com uma mensagem para um nome de sincronização (por exemplo, c) passado pelo invocador se está ou não activado — no caso de não estar, essa mensagem é $c \lhd \mathit{off}$, e se está, a mensagem é $c \lhd \mathit{out} : i$, onde i é um novo nome, que denota um objecto que disponibiliza o método a executar;
- se estava activado, o invocador envia, ao objecto que disponibiliza o método, uma confirmação da execução: então realizam-se as invocações de serviços (sequencialmente, para consumir as mensagens de sincronização) e executam-se depois a valoração e o retorno de valores, em paralelo; deve ser enviada a mensagem $c \lhd \mathit{out}$ ao invocador, eventualmente com um valor de retorno, e deve ser reposto o objecto que representa a instância, eventualmente alterando o valor passado no parâmetro que denota o atributo da classe. Pode também o invocador enviar uma ordem para desistir da execução do serviço.

Veja-se, com um outro exemplo, como fica agora o agente que representa a classe:

Exemplo 4.2.2 *Semáforo*

$[\![$`class` Sem
  `interface`
    `serv` *req*
    `serv` *rel*
  `body`
    `slot` s: `nat`
    `serv` *req*
      `enb` $s > 0$
      `val` $s \ll s - 1$
    `serv` *rel*
      `enb` $s < 1$
      `val` $s \ll s + 1$
`end` Sem$]\!] =$

$$
\begin{aligned}
\texttt{rec}\ \mathrm{Sem}\ &=\ (\mathit{self}\, s)\ \mathit{self} \triangleright [\, \mathit{req} : (c)\, \texttt{if}\ [\![s > 0]\!]\ \texttt{then}\ \nu t\ c \triangleleft \mathit{out} : t, \mathrm{Req}(t\ \mathit{self}\, s) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else}\ c \triangleleft \mathit{off},\ \mathrm{Sem}(\mathit{self}\, s)\ \texttt{fi}\ \& \\
&\qquad\qquad\qquad\qquad \mathit{rel} : (c)\, \texttt{if}\ [\![s < 1]\!]\ \texttt{then}\ \nu t\ c \triangleleft \mathit{out} : t, \mathrm{Rel}(t\ \mathit{self}\, s) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else}\ c \triangleleft \mathit{off},\ \mathrm{Sem}(\mathit{self}\, s)\ \texttt{fi}\ ]
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Req}\ &=\ (\mathit{self}\ i\ s)\ \mathit{self} \triangleright [\, \mathit{commit} : (c)\ c \triangleleft \mathit{out}, \mathrm{Sem}(i\ [\![s - 1]\!])\ \& \\
&\qquad\qquad\qquad\qquad \mathit{abort} :\ \ (c)\ c \triangleleft \mathit{out},\ \mathrm{Sem}(i\ s)\ ]
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Rel}\ &=\ (\mathit{self}\ i\ s)\ \mathit{self} \triangleright [\, \mathit{commit} : (c)\ c \triangleleft \mathit{out}, \mathrm{Sem}(i\ [\![s + 1]\!])\ \& \\
&\qquad\qquad\qquad\qquad \mathit{abort} :\ \ (c)\ c \triangleleft \mathit{out},\ \mathrm{Sem}(i\ s)\ ]
\end{aligned}
$$

■

Logo, uma instância tem dois estados, entre os quais alterna: o estado *livre*, em que disponibiliza os seus serviços (*req* e *rel*, neste caso); o estado *bloqueado*, em que recebeu o pedido de execução de um serviço (que estava activado) e está à espera da ordem de efectivação da execução (*commit*) ou de desistência (*abort*). Não houve ainda mudanças de estado e não pode realizar qualquer outra acção.

No estado livre, a instância permite avaliar a activação dos seus serviços; no estado bloqueado permite apenas que o serviço pedido (que estava activado) possa ser executado ou cancelado, não sendo possivel realizar nenhuma outra acção. Por isso, no estado livre a instância é representada por um objecto que tem como métodos os serviços da classe e no estado bloqueado a instância é representada por um objecto com dois métodos, um para a opção "commit", que se compromete com a execução do serviço pedido, e outro para a opção "abort", que permite desistir da execução do serviço, não chegando a haver mudanças de estado. Após ser dada uma destas ordens, a instância volta ao estado livre.

Esta implementação da execução de um serviço, em dois tempos, emula um protocolo do tipo "commit-abort", como é proposto em [Tel 94].

Incorporando este protocolo, o agente que representa a classe passa a ser:

$$
\begin{aligned}
&\texttt{rec}\ \mathrm{Class} = (\mathit{self}\ \mathit{id\_slot})\\
&\qquad \mathit{self} \triangleright [\mathit{id\_serv} : (\mathit{id\_par}\ r)\ \texttt{if}\ [\![\mathbf{condição}]\!]\ \texttt{then}\\
&\qquad\qquad \nu t\ r \triangleleft \mathit{out} : t, \mathrm{CommitAbort}(t\ \mathit{self}\ \mathit{id\_slot})\\
&\qquad\quad \texttt{else}\\
&\qquad\qquad r \triangleleft \mathit{off}, \mathrm{Class}(\mathit{self}\ \mathit{id\_slot})\\
&\qquad\quad \texttt{fi}]
\end{aligned}
$$

onde o agente CommitAbort é:

$$
\begin{aligned}
&\mathrm{CommitAbort} = (\mathit{self}\ c\ \mathit{id\_slot})\\
&\qquad \mathit{self} \triangleright [\ \mathit{commit} : (r)\ r \triangleleft \mathit{out} : [\![\mathbf{exp\_ret}]\!], \mathrm{Class}(c\ [\![\mathbf{exp\_val}]\!]))\ \&\\
&\qquad\qquad \mathit{abort} : (r)\ r \triangleleft \mathit{off}, \mathrm{Class}(c\ \mathit{id\_slot})]
\end{aligned}
$$

O agente Class contém, no corpo do método, um processo que representa a instância no estado bloqueado; esse processo resulta da aplicação dos nomes t *self id_slot* ao agente CommitAbort. O nome t identifica a instância no estado bloqueado, *self* é o nome da instância no estado livre e *id_slot* é o atributo, que pode ser modificado pela expressão de valoração se o serviço for executado (opção *commit*)

Invocação de serviços

Esta construção constitui o mecanismo de chamada de acções externas de objectos GNOME: permite que o "mundo exterior" comunique com uma instância de uma classe GNOME de forma síncrona, solicitando-lhe um serviço (que será executado atomicamente).

Para se traduzir uma chamada de determinado serviço, uma vez que a comunicação em TyCO é assíncrona, têm que se considerar duas fases distintas na sua execução, como se disse atrás: a avaliação da sua activação e a sua (eventual) execução. A instância da classe que disponibiliza o serviço em questão estará inicialmente no estado livre; no caso do serviço invocado estar activado, passa ao estado bloqueada, ficando à espera de uma ordem de comprometimento com a execução (*commit*) ou de desistencia (*abort*); após receber uma das duas, volta ao estado livre. Desta forma emula-se a atomicidade da execução do serviço e a comunicação síncrona.

Apresenta-se um exemplo de uma classe com interacção; usam-se dois atributos (excepção ao definido inicialmente), servindo o segundo para guardar a identidade da instância chamada.

Exemplo 4.2.3 *Cliente de semáforo*

```
[[class Cliente
  interface
    serv noncrit
    serv request
    serv crit
    serv release
  body
    slot st: [nc,rq,cr,rl]
    slot s: Sem
    serv noncrit
      enb st = nc
```

```
    val st ≪ rq
  serv request
    enb st = rq
    call req of s: Sem
    val st ≪ cr
  serv crit
    enb st = cr
    val st ≪ rl
  serv release
    enb st = rl
    call rel of s: Sem
    val st ≪ nc
end Cliente⟧ =
```

$$
\begin{array}{lcl}
\texttt{rec}\ \mathrm{C} & = & (\mathit{self}\,\mathit{st}\ s)\\
& & \quad \mathit{self} \triangleright [\ \mathit{noncrit} : (r)\,\texttt{if}\ [\![\mathit{st} = \texttt{nc}]\!]\ \texttt{then}\ \nu t\ r \triangleleft \mathit{out} : t, \mathrm{Nct}(t\ \mathit{self}\,\mathit{st}\ s)\\
& & \qquad\qquad\qquad\qquad \texttt{else}\ r \triangleleft \mathit{off}, \mathrm{C}(\mathit{self}\,\mathit{st}\ s)\ \texttt{fi}\ \&\\
& & \qquad\quad \mathit{request} : (r)\,\texttt{if}\ [\![\mathit{st} = \texttt{rq}]\!]\ \texttt{then}\\
& & \qquad\qquad\qquad\qquad\quad \texttt{case}\ s \triangleleft \mathit{req}\ \texttt{of}\\
& & \qquad\qquad\qquad\qquad\qquad \mathit{out} : (v)\ \nu t\ r \triangleleft \mathit{out} : t, \mathrm{Rqt}(t\ v\ \mathit{self}\,\mathit{st}\ s)\\
& & \qquad\qquad\qquad\qquad\qquad \mathit{off} : r \triangleleft \mathit{off}, \mathrm{C}(\mathit{self}\,\mathit{st}\ s)\\
& & \qquad\qquad\qquad\qquad\quad \texttt{esac}\\
& & \qquad\qquad\qquad\qquad \texttt{else}\ r \triangleleft \mathit{off}, \mathrm{C}(\mathit{self}\,\mathit{st}\ s)\ \texttt{fi}\ \&\\
& & \qquad\quad \mathit{crit} : \quad (r)\,\texttt{if}\ [\![\mathit{st} = \texttt{cr}]\!]\ \texttt{then}\ \nu t\ r \triangleleft \mathit{out} : t, \mathrm{Crt}(t\ \mathit{self}\,\mathit{st}\ s)\\
& & \qquad\qquad\qquad\qquad \texttt{else}\ r \triangleleft \mathit{off}, \mathrm{C}(\mathit{self}\,\mathit{st}\ s)\ \texttt{fi}\ \&\\
& & \qquad\quad \mathit{release} : (r)\ \texttt{if}\ [\![\mathit{st} = \texttt{rl}]\!]\ \texttt{then}\\
& & \qquad\qquad\qquad\qquad\quad \texttt{case}\ s \triangleleft \mathit{rel}\ \texttt{of}\\
& & \qquad\qquad\qquad\qquad\qquad \mathit{out} : (v)\ \nu t\ r \triangleleft \mathit{out} : t, \mathrm{Rls}(t\ v\ \mathit{self}\,\mathit{st}\ s)\\
& & \qquad\qquad\qquad\qquad\qquad \mathit{off} : r \triangleleft \mathit{off}, \mathrm{C}(\mathit{self}\,\mathit{st}\ s)\\
& & \qquad\qquad\qquad\qquad\quad \texttt{esac}\\
& & \qquad\qquad\qquad\qquad \texttt{else}\ r \triangleleft \mathit{off}, \mathrm{C}(\mathit{self}\,\mathit{st}\ s)\ \texttt{fi}\ ]\\
\mathrm{Nct} & = & (\mathit{self}\,c\ \mathit{st}\ s)\ \mathit{self} \triangleright [\ \mathit{commit} : (r)\ r \triangleleft \mathit{out}, \mathrm{C}(c\ [\![\texttt{rq}]\!]\ s)\ \&\\
& & \qquad\qquad\qquad\qquad \mathit{abort} : (r)\ r \triangleleft \mathit{out}, \mathrm{C}(c\ \mathit{st}\ s)\ ]\\
\mathrm{Rqt} & = & (\mathit{self}\,v\ c\ \mathit{st}\ s)\ \mathit{self} \triangleright [\ \mathit{commit} : (r)\ v \triangleleft \mathit{commit}; (r \triangleleft \mathit{out}, \mathrm{C}(c\ [\![\texttt{cr}]\!]\ s))\ \&\\
& & \qquad\qquad\qquad\qquad\quad \mathit{abort} : (r)\ v \triangleleft \mathit{abort}; (r \triangleleft \mathit{out}, \mathrm{C}(c\ \mathit{st}\ s))\ ]\\
\mathrm{Crt} & = & (\mathit{self}\,c\ \mathit{st}\ s)\ \mathit{self} \triangleright [\ \mathit{commit} : (r)\ r \triangleleft \mathit{out}, \mathrm{C}(c\ [\![\texttt{rl}]\!]\ s)\ \&\\
& & \qquad\qquad\qquad\qquad \mathit{abort} : (r)\ r \triangleleft \mathit{out}, \mathrm{C}(c\ \mathit{st}\ s)\ ]\\
\mathrm{Rls} & = & (\mathit{self}\,v\ c\ \mathit{st}\ s)\ \mathit{self} \triangleright [\ \mathit{commit} : (r)\ v \triangleleft \mathit{commit}; (r \triangleleft \mathit{out}, \mathrm{C}(c\ [\![\texttt{nc}]\!]\ s))\ \&\\
& & \qquad\qquad\qquad\qquad\quad \mathit{abort} : (r)\ v \triangleleft \mathit{abort}; (r \triangleleft \mathit{out}, \mathrm{C}(c\ \mathit{st}\ s))\ ]
\end{array}
$$

■

No caso em que se invoca o serviço *request* ou *release*, há que considerar que será, por sua vez invocado, o serviço *req* ou *rel* de uma instância do semáforo. Por isso, no método *commit* respectivo, usa-se a sincronização de processos[8], de forma a que só se realize a

[8] Definida em 3.1.13.

valoração após ser recebida a indicação que terminou com sucesso a execução do serviço invocado.

Como se faz a interacção entre uma instância do agente Cliente de Semáforo e uma instância do agente Semáforo? Veja-se operacionalmente o que se passa se houver um pedido de execução do serviço *request*, sendo i uma instância da classe Cliente e s uma instância da classe Sem; suponha-se que:

1. O Cliente do Semáforo não está no estado correcto.

$$\mathrm{C}(i\ nc\ s), i \lhd request : d \twoheadrightarrow d \lhd \mathit{off}, \mathrm{C}(i\ nc\ s)$$

   Responde que não está activado, não alterando o seu estado; não houve interacção.

2. O Semáforo não está disponível.
   $\mathrm{Sem}(s\ 0), \mathrm{C}(i\ rq\ s), i \lhd request : d \twoheadrightarrow$
   $\nu c\ \mathrm{Sem}(s\ 0), s \lhd req : c,$
   $c \rhd [out : (v)\ \nu q\ d \lhd out : q, \mathrm{Rqt}(q\ i\ v\ st\ s)\ \&\ \mathit{off} : d \lhd \mathit{off}, \mathrm{C}(i\ rq\ s)\ ] \twoheadrightarrow$
   $\nu c\ \mathrm{Sem}(s\ 0), c \lhd \mathit{off},$
   $c \rhd [out : (v)\ \nu q\ d \lhd out : q, \mathrm{Rqt}(q\ i\ v\ st\ s)\ \&\ \mathit{off} : d \lhd \mathit{off}, \mathrm{C}(i\ rq\ s)\ ] \rightarrow$
   $\mathrm{Sem}(s\ 0), \mathrm{C}(i\ rq\ s), d \lhd \mathit{off}$

   Responde que não está activado, não alterando o seu estado; houve um pedido ao semáforo do serviço *req* mas este não estava activado.

3. Reunem-se as condições para a interacção.
   $\mathrm{Sem}(s\ 1), \mathrm{C}(i\ rq\ s), i \lhd request : d \twoheadrightarrow$
   $\nu c\ \mathrm{Sem}(s\ 1), s \lhd req : c,$
   $c \rhd [out : (v)\ \nu t\ d \lhd out : t, \mathrm{Rqt}(t\ v\ i\ st\ s)\ \&\ \mathit{off} : d \lhd \mathit{off}, \mathrm{C}(i\ rq\ s)\ ] \twoheadrightarrow$
   $\nu c\ e\ c \lhd out : e, \mathrm{Req}(e\ s\ 1),$
   $c \rhd [out : (v)\ \nu t\ d \lhd out : t, \mathrm{Rqt}(t\ v\ i\ st\ s)\ \&\ \mathit{off} : d \lhd \mathit{off}, \mathrm{C}(i\ rq\ s)\ ] \rightarrow$
   $\nu e\ t\ \mathrm{Req}(e\ s\ 1), d \lhd out : t, \mathrm{Rqt}(t\ i\ e\ st\ s)$

   Ao invocador é enviado o nome do objecto (t) que tem o método *request*. Se vem a confirmação da execução, $t \lhd commit : d$, prossegue a computação, enviando-se a ordem *commit* ao semáforo:

   $\nu e\ t\ q \lhd commit : d, \mathrm{Req}(e\ s\ 1), \mathrm{Rqt}(t\ e\ st\ s) \twoheadrightarrow$
   $\nu c\ e\ \mathrm{Req}(e\ s\ 1), e \lhd commit : c, c \rhd [out : c \lhd out, \mathrm{C}(i\ cr\ s)\ ] \twoheadrightarrow$
   $\nu c\ \mathrm{Sem}(s\ 0), c \lhd out, c \rhd [out : d \lhd out, \mathrm{C}(i\ cr\ s)\ ] \rightarrow$
   $\mathrm{Sem}(s\ 0), \mathrm{C}(i\ cr\ s), d \lhd out$

   Altera o valor do semáforo e do estado do cliente.

Nova codificação para a classe

Apresentado o exemplo importa analisar os requisitos da tradução da invocação de serviços. Recorde-se a gramática de uma cláusula de chamada:

`call` *id_serv* [`as` *id_serv*] `of` **expressão_chamada** : *id_class*
    (`arg` *id_serv.id_par* = **expressão**)*

Como já se disse, quando se invoca um serviço realizam-se várias tarefas, em duas fases distintas: a avaliação da activação desse serviço e a sua eventual execução.Seguidamente explica-se, em detalhe, cada uma dessas fases.

Na primeira fase, começa-se por avaliar **expressão_chamada**, de forma a identificar a instância invocada, e avaliar as várias expressões (**expressão**), de forma a instanciar os parâmetros do serviço. Depois, envia-se ao objecto que identifica a instância uma mensagem com a identidade do serviço chamado (*id_serv*), passando os nomes que representam os parâmetros desse serviço e um nome (por exemplo, d), recém-criado, onde espera receber os resultados. Este último nome é utilizado para sincronização de processos, pois simula-se a comunicação síncrona com duas comunicações assíncronas, em sentidos inversos: o invocador envia um pedido de execução e fica bloqueado à espera de receber uma resposta que lhe permita prosseguir a sua acção.

Na segunda fase, se o serviço invocado não está activado o invocador recebe a resposta $d \lhd \mathit{off}$, e, por consequencia, a acção que contém a chamada também não estará activada. Não se pode prosseguir a execução da acção; então restaura-se o objecto que identifica a instância, sem alterar o valor do atributo, e responde-se que a acção não está activada, com uma mensagem $c \lhd \mathit{off}$ (c é um nome passado pelo invocador). Se o serviço invocado está activado o invocador recebe uma mensagem $c \lhd \mathit{out} : v$, onde v é um nome novo que identifica um objecto com os métodos *commit* (que permite ao invocador executar o serviço) e *abort* (que permite desistir da sua execução). Em ambos os métodos, a sua execução implica enviar, ao invocador do serviço, a mensagem $c \lhd \mathit{out}$ indicando que terminou a execução do método (e, eventualmente, passando os resultados), e repõe-se o objecto que representa a instância no estado livre, terminando a interacção entre os dois objectos.

Para generalizar a tradução da invocação de serviços, uma vez que no corpo de uma acção podem surgir várias cláusulas de chamada e que devem estar todas activadas para que a acção esteja activada, apresenta-se a tradução de uma classe com um serviço com duas cláusulas de chamada (que é já suficientemente geral).

⟦`class` Class
  `interface`
    `serv` *id_serv*
      `par` *id_par*: *id_tipo*
      `res` *id_res*: *id_tipo*
  `body`
    `slot` *id_slot*: *id_tipo*
    `serv` *id_serv*
      `enb` **condição**
      `call` v_1 `of` **alvo_1**: C_1
        `arg` v_1.*id_par* = **arg_1**
      `call` v_2 `of` **alvo_2**: C_2
        `arg` v_2.*id_par* = **arg_2**
      `val` *id_slot* ≪ **exp_val**
      `ret` *id_res* = **exp_ret**
`end` Class⟧ =

$$
\begin{array}{l}
\texttt{rec}\ \mathrm{Class} = (\mathit{self}\ \mathit{id\_slot}) \\
\qquad \mathit{self} \rhd [\ \mathit{id\_serv} : (\mathit{id\_par}\ r)\ \texttt{if}\ [\![\mathbf{condição}]\!]\ \texttt{then} \\
\qquad\qquad\qquad \nu x_1\ [\![\mathbf{alvo\_1}]\!] \lhd v_1 : [\![\mathbf{arg\_1}]\!]\ x_1, \\
\qquad\qquad\qquad\quad x_1 \rhd [\ \mathit{out} : (v_1)\ \nu x_2\ [\![\mathbf{alvo\_2}]\!] \lhd v_2 : [\![\mathbf{arg\_2}]\!]\ x_2, \\
\qquad\qquad\qquad\qquad\qquad x_2 \rhd [\ \mathit{out} : \mathrm{Commit}\ \&\ \mathit{off} : \mathrm{Abort}\ ]\ \& \\
\qquad\qquad\qquad\qquad \mathit{off} : r \lhd \mathit{off}, \mathrm{Class}(\mathit{self}\ \mathit{id\_slot})\ ] \\
\qquad\qquad\quad \texttt{else} \\
\qquad\qquad\qquad r \lhd \mathit{off}, \mathrm{Class}(\mathit{self}\ \mathit{id\_slot}) \\
\qquad\qquad\quad \texttt{fi}]
\end{array}
$$

onde os processos Commit e Abort são, respectivamente:

$$\mathrm{Commit} = (v_2)\ \nu t\ r \lhd \mathit{out} : t, \mathrm{CommitAbort}(t\ \mathit{self}\ v_1\ v_2\ \mathit{id\_slot}\ \mathit{id\_par})$$

$$\mathrm{Abort} = v_1 \lhd \mathit{abort}\ ; (r \lhd \mathit{off}, \mathrm{Class}(\mathit{self}\ \mathit{id\_slot}))$$

$$
\begin{array}{l}
\mathrm{CommitAbort} = (\mathit{self}\ v_1\ v_2\ c\ \mathit{id\_slot}\ \mathit{id\_par}) \\
\qquad \mathit{self} \rhd [\ \mathit{commit} : (r)\ v_1 \lhd \mathit{commit} : [\![\mathbf{arg\_1}]\!]\ ; v_2 \lhd \mathit{commit} : [\![\mathbf{arg\_2}]\!]\ ; \\
\qquad\qquad\qquad (r \lhd \mathit{out} : [\![\mathbf{exp\_ret}]\!], \mathrm{Class}(c\ [\![\mathbf{exp\_val}]\!]))\ \& \\
\qquad\quad \mathit{abort} : (r)\ v_1 \lhd \mathit{abort}\ ; v_2 \lhd \mathit{abort}\ ; (r \lhd \mathit{out}, \mathrm{Class}(c\ \mathit{id\_slot}))]
\end{array}
$$

O método do agente Class pretende fazer duas chamadas a serviços; para tal, avalia a activação do primeiro e, se verifica que ele está activado, avalia então a activação do segundo; se algum deles não está activado, o serviço da classe também não o estará, devendo isso ser comunicado ao invocador - envia-se a mensagem *off* ao nome passado no parâmetro r — e não se pode fazer mais nada (repõe-se a instância no estado livre). Se é o segundo serviço chamado que não está activado é necessário desistir da execução do primeiro serviço invocado, passando a instância que o disponibiliza ao estado livre (invoca-se o agente Abort); se ambos os serviços invocados estão activados pode ser executado todo o serviço da classe (invoca-se o agente Commit), devendo a instância passar ao estado bloqueado: cria um objecto com os métodos *commit* e *abort* (instância de CommitAbort). Qualquer um destes métodos envia sequencialmente as ordens (de comprometimento ou desistência) aos serviços invocados e depois, em paralelo, envia uma mensagem ao invocador, indicando que terminou a execução, e finalmente repõe a instância no estado livre.

Usaram-se as duas formas de sincronização de processos definidas em 3.1.13 para que a instância tenha um comportamento que emule a execução atómica de um método. Mas isso não acontece totalmente: a execução sequencial das chamadas justifica-se apenas pela necessidade de "consumir" as mensagens de sincronização, sendo uma limitação ao paralelismo existente na execução de um método em Gnome.

4.3 Criação dinâmica de objectos

Até aqui pressupôs-se que se estão a pedir serviços a instâncias existentes. Diz a semântica do Gnome que, se é pedido um serviço a uma instância que não existe, a resposta é que esse serviço não está activado. Mas não é este o comportamento do objecto implementado. Se se enviar um pedido de serviço a um nome que não corresponde a nenhum objecto, devido à sincronização, o invocador vai ficar sempre à espera da ordem para prosseguir, ficando numa situação de impasse. Para resolver esta questão os objectos devem ser explicitamente criados (e "mortos"), e deve-se manter uma lista das instâncias existentes para uma dada classe.

4.3.1 Serviços de nascimento e de morte

Quer-se que seja possível saber se uma dada instância existente ou não. Mais, é necessário também poder criar e "matar" instâncias, uma vez que o GNOME suporta a criação e morte dinâmica de objectos.

Por inspiração na linguagem Smalltalk [LaLonde e Pugh 90], onde as classes têm métodos, tal como os objectos, para realizar as tarefas referidas basta ter um agente com os métodos para a criação, morte e verificação da existência de instâncias, mais os métodos da classe GNOME.

Refinando esta solução, pode-se separar os métodos para dois objectos (instâncias de agentes): um com os métodos de nascimento, morte e verificação da existência de instâncias da classe e outro que representa uma dada instância (disponibilizando os seus métodos).

Mas para se saber se dada instância existe ou não, é preciso manter um registo das instâncias que se vão criando (e actualiza-lo quando se mata uma instância). Associa-se a cada classe um conjunto[9] que guarda os identificadores das instâncias existentes, e um objecto — o gestor da classe — com três métodos: um para o nascimento de instâncias, um que permite verificar se determinada instância existe ou não, e um para matar instâncias.

Uma vez que as classes estão inseridas no contexto de uma especificação, sendo os serviços de cada uma acessíveis às outras classes dessa especificação, o nome que denota o objecto que representa o gestor de uma classe (g, por exemplo) deve ser visível para outros objectos que representem os gestores das outras classes da especificação. O identificador do conjunto associado a cada classe só dela deve ser conhecido (é-lhe local e, por isso, deve estar capsulado do exterior).

Para ilustrar a codificação da classe que resulta das considerações feitas, apresenta-se uma nova versão do exemplo 4.2.1 (um contador), agora com um serviço de nascimento e outro de morte:

Exemplo 4.3.1 *Contador*

```
⟦class Cont
 interface
  birth serv new
  death serv end
  serv inc
  serv set
   par p: nat
  serv obs
   res r: nat
 body
```

[9] De acordo com a definição 3.1.11.

```
  slot n: nat
  serv new
    val n ≪ 0
  serv inc
    val n ≪ n + 1
  serv set
    val n ≪ set.p
  serv obs
    ret obs.r = n
  serv end
    enb n ≥ 0
end Cont]] =
```

$[\![$`user class` Cont $\cdots$ `end` Cont$]\!] = (g)\,\nu l\ \mathrm{Class}(g\ l), \mathrm{Novo}(l)$

$$
\begin{aligned}
\texttt{rec}\ \mathrm{Class} &= (\mathit{self}\, l)\\
&\quad \mathit{self} \triangleright [\,\mathit{birth} : (r)\ \nu n\ r \lhd \mathit{out} : n, \mathrm{New}(n\ \mathit{self}\, l), \mathrm{Class}(\mathit{self}\, l)\ \&\\
&\qquad\quad \mathit{death} : (i\ r)\ l \lhd \mathit{retirar} : i\,;(r \lhd \mathit{out}, \mathrm{Class}(\mathit{self}\, l))\ \&\\
&\qquad\quad \mathit{existe} : (i\ r)\ \texttt{if}\ l \lhd \mathit{existe} : i\ \texttt{then}\ r \lhd \mathit{out}, \mathrm{Class}(\mathit{self}\, l)\\
&\qquad\qquad\qquad \texttt{else}\ r \lhd \mathit{off}, \mathrm{Class}(\mathit{self}\, l)\ \texttt{fi}\,]
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{New} &= (\mathit{self}\, g\ l)\ \mathit{self} \triangleright [\,\mathit{commit} : (r)\ \nu i\ l \lhd \mathit{inserir} : i\,;(r \lhd \mathit{out} : i, \mathrm{Cont}(i\ g\ [\![0]\!]))\ \&\\
&\qquad \mathit{abort} : (r)\ r \lhd \mathit{out}\,]
\end{aligned}
$$

$$
\begin{aligned}
\texttt{rec}\ \mathrm{Cont} &= (\mathit{self}\, g\ n)\\
&\quad \mathit{self} \triangleright [\,\mathit{inc} : (r)\ \nu t\ r \lhd \mathit{out} : t, \mathrm{Inc}(t\ \mathit{self}\, g\ n)\ \&\\
&\qquad\quad \mathit{set} : (p\ r)\ \nu t\ r \lhd \mathit{out} : t, \mathrm{Set}(t\ \mathit{self}\, g\ p\ n)\ \&\\
&\qquad\quad \mathit{obs} : (r)\ \nu t\ r \lhd \mathit{out} : t, \mathrm{Obs}(t\ \mathit{self}\, g\ n)\ \&\\
&\qquad\quad \mathit{end} : (r)\ \texttt{if}\ [\![n \geq 0]\!]\ \texttt{then}\ \nu t\ r \lhd \mathit{out} : t, \mathrm{End}(t\ \mathit{self}\, g\ n)\\
&\qquad\qquad \texttt{else}\ r \lhd \mathit{off}\ \texttt{fi}, \mathrm{Cont}(\mathit{self}\, g\ n)\,]
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Inc} &= (\mathit{self}\, c\ g\ n)\ \mathit{self} \triangleright [\,\mathit{commit} : (r)\ r \lhd \mathit{out}, \mathrm{Cont}(c\ g\ [\![n+1]\!])\ \&\\
&\qquad \mathit{abort} : (r)\ r \lhd \mathit{out}, \mathrm{Cont}(c\ g\ n)\,]
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Set} &= (\mathit{self}\, c\ g\ p\ n)\ \mathit{self} \triangleright [\,\mathit{commit} : (r)\ r \lhd \mathit{out}, \mathrm{Cont}(c\ g\ p)\ \&\\
&\qquad \mathit{abort} : (r)\ r \lhd \mathit{out}, \mathrm{Cont}(c\ g\ n)\,]
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Obs} &= (\mathit{self}\, c\ g\ n)\ \mathit{self} \triangleright [\,\mathit{commit} : (r)\ r \lhd \mathit{out} : n, \mathrm{Cont}(c\ g\ n)\ \&\\
&\qquad \mathit{abort} : (r)\ r \lhd \mathit{out}, \mathrm{Cont}(c\ g\ n)\,]
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{End} &= (\mathit{self}\, c\ g\ n)\ \mathit{self} \triangleright [\,\mathit{commit} : (r)\ g \lhd \mathit{death} : c\,; r \lhd \mathit{out}\ \&\\
&\qquad \mathit{abort} : (r)\ r \lhd \mathit{out}, \mathrm{Cont}(c\ g\ n)\,]
\end{aligned}
$$

■

Esclarecem-se apenas mais dois aspectos desta nova representação da classe:

- após ser "criada" a instância, a sua identidade é transmitida ao invocador na mensagem de sincronização e colocada no conjunto das instâncias da classe. Esse nome denota também uma instância do agente Cont, que disponibiliza os serviços da classe (excepto o de nascimento, como já se viu), que representa a instância no estado livre.
- ao ser confirmada a execução do serviço de morte, requere-se ao gestor da classe que

retire a identidade da instância do conjunto das instâncias da classe, e não se repõe o objecto que disponibiliza os serviços, desaparecendo assim do ambiente a instância.

Para ver o aspecto do agente que representa a classe típica é necessário fazer uma nova tradução do mecanismo de invocação de serviços, considerando a existência do gestor da classe.

4.3.2 Nova codificação da chamada

A introdução do gestor das instâncias de uma classe tem influência na tradução das cláusulas de chamada: agora, antes de invocar um serviço de determinada instância, é necessário ver se ela existe.

A classe com estrutura típica, com um serviço de nascimento e outro de morte, e com a nova codificação da chamada, e agora representada pelo agente Class, que se apresenta a seguir. Uma sua instância é o gestor das instâncias da classe e considera-se que existe apenas um por classe, no âmbito de uma especificação. Este gestor é um objecto que tem três métodos: *birth*, que permite criar instâncias da classe; *death*, que "mata" instâncias da classe; *existe*, que verifica se existe uma determinada instância da classe.

Apresenta-se, a seguir, a nova codificação da classe com estrutura típica, supondo que os serviços têm chamada:

$[\![\texttt{class}\ \mathrm{C} \cdots \texttt{end}\ \mathrm{C}]\!] = (g)\ \nu l\ \mathrm{Class}(g\ l), \mathrm{Novo}(l)$

$$
\begin{array}{l}
\mathrm{Class} = (\mathit{self}\, l)\ \texttt{rec}\ \mathrm{Class} \\
\quad \mathit{self} \rhd [\ \mathit{birth} : (\mathit{id\_par}\ r)\, \texttt{if}\ [\![\mathbf{condição}]\!]\ \texttt{then} \\
\qquad\qquad\qquad \texttt{if}\ g \lhd \mathit{existe} : [\![\mathbf{alvo\_1}]\!]_{i_1}\ \texttt{then} \\
\qquad\qquad\qquad\quad i_1 \lhd v_1 : [\![\mathbf{arg\_1}]\!]_{g_1}\ ; \\
\qquad\qquad\qquad\quad (\texttt{if}\ g \lhd \mathit{existe} : [\![\mathbf{alvo\_2}]\!]_{i_2}\ \texttt{then} \\
\qquad\qquad\qquad\qquad i_2 \lhd v_2 : [\![\mathbf{arg\_2}]\!]_{g_2}\ ; (\mathrm{Commit} \mid \mathrm{Abort}) \\
\qquad\qquad\qquad\quad\ \texttt{else} \\
\qquad\qquad\qquad\qquad \mathrm{Abort} \\
\qquad\qquad\qquad\quad\ \texttt{fi}) \\
\qquad\qquad\qquad\quad \mid r \lhd \mathit{off}, \mathrm{Class}(\mathit{self}\, l)) \\
\qquad\qquad\qquad \texttt{else} \\
\qquad\qquad\qquad\quad r \lhd \mathit{off}, \mathrm{Class}(\mathit{self}\, l) \\
\qquad\qquad\qquad \texttt{fi} \\
\qquad\qquad\quad \texttt{else} \\
\qquad\qquad\qquad r \lhd \mathit{off}, \mathrm{Class}(\mathit{self}\, l) \\
\qquad\qquad\quad \texttt{fi}\ \& \\
\qquad \mathit{death} : (i\ r)\ l \lhd \mathit{retirar} : i\ ; (r \lhd \mathit{out}, \mathrm{Class})\ \& \\
\qquad \mathit{existe} : (i\ r)\ \texttt{if}\ l \lhd \mathit{existe} : i\ \texttt{then}\ r \lhd \mathit{out}, \mathrm{Class} \\
\qquad\qquad\quad \texttt{else}\ r \lhd \mathit{off}, \mathrm{Class}\ \texttt{fi}\ ]
\end{array}
$$

onde os processos Commit e Abort são, respectivamente:

$$\mathrm{Commit} \ = \ \nu n\ r \lhd \mathit{out} : n, \mathrm{New}(n\ v_1\ v_2\ g_1\ g_2\ \mathit{self}\, g\ \mathit{id\_par}\ l)$$

$$\mathrm{Abort} \ = \ v_1 \lhd \mathit{abort}\ ; (c \lhd \mathit{off}, \mathrm{Class}(\mathit{self}\, l))$$

$$\begin{aligned}
\mathrm{New} = (&\mathit{self}\ v_1\ v_2\ g_1\ g_2\ c\ g\ \mathit{id_par}\ l) \\
&\mathit{self} \rhd [\ \mathit{commit} : (r)\ \nu i\ l \lhd \mathit{inserir} : i\ ; (i_1 \lhd \mathit{commit} : g_1\ ; i_2 \lhd \mathit{commit} : g_2\ ; \\
&\qquad (r \lhd \mathit{out} : [\![\mathbf{exp_ret}]\!], \mathrm{Serv}(i\ g\ [\![\mathbf{exp_val}]\!]))) \ \& \\
&\quad \mathit{abort} : (r)\ v_1 \lhd \mathit{abort}\ ; v_2 \lhd \mathit{abort}\ ; (r \lhd \mathit{out}, \mathrm{Class}(\mathit{self}\ l))]
\end{aligned}$$

No corpo de *birth*, verifica-se a activação do serviço de nascimento (depende dum eventual parâmetro e de eventuais chamadas) e, no caso de estar activado, é criado um objecto (instância de New) com os métodos *commit* (executa as restantes cláusulas do método do serviço de nascimento) e *abort* (que permite desistir da criação da nova instância). É este objecto que cria de facto as instâncias da classe: cria um novo nome para denotar um objecto (instância do agente Serv) que tem, como métodos, os restantes serviços da classe; o seu nome é colocado no conjunto, pois passa a ser a identidade da nova instância.

Quando se invoca o serviço de morte de uma instância, ela envia uma mensagem (*death*) ao gestor da classe, para que este retire a identificação da instância do conjunto de instâncias da classe.

Para invocar serviços de instâncias de determinada classe deve-se sempre verificar primeiro se elas existem: invoca-se o serviço *existe* do gestor da classe, passando-lhe o nome da instância a procurar (no conjunto das instâncias da classe) e o nome onde espera a resposta.

Relativamente às invocações de serviços, observe-se que agora se optou por fazer apenas uma vez a avaliação das expressões: os nomes que guardam os resultados são passados nos parâmetros dos agentes cujas instâncias representam as instâncias da classe, no estado bloqueado.

O agente Serv é semelhante ao agente Class apresentado no final da secção anterior, mas tem mais um parâmetro: a identidade do gestor da classe. O seu atributo é inicializado com o resultado da avaliação da expressão de valoração que surge no seu serviço de nascimento. Este agente tem como métodos os serviços da classe (excepto o de nascimento), onde o corpo de cada um dos seus métodos tem a mesma estrutura do corpo do método *birth*.

No caso de se executar o serviço de morte, é enviada a mensagem com etiqueta *death* ao gestor (para que este retire a identidade da instância do conjunto), e não se repoe o objecto que disponibiliza os serviços,desaparecendo assim a instância do "ambiente".

4.4 Classes de objectos não deterministas

4.4.1 Serviços com alternativas

O Gnome permite que as acções tenham alternativas[10]. A semântica dita que, de entre as que estão activadas, é escolhida, de forma não determinista, uma para ser executada, e se não estiver nenhuma activada, o serviço também não o estará. É requerida justiça forte[11] na gestão das alternativas.

[10] Ver capítulo 2

[11] Se determinada acção está recorrentemente activada tem que acabar por ser executada [Manna e Pnueli 92].

Para implementar esta semântica uma instância vai testar rotativamente as alternativas de cada serviço, de forma a dar iguais oportunidades a cada uma delas; a primeira a ser testada é a seguinte à ultima que foi executada (a primeira, no momento inicial), que, se está activada é executada, senão passa-se à próxima. O procedimento repete-se até se encontrar alguma activada ou até se ter testado todas não encontrando nenhuma activada, e neste caso o serviço também não está activado.

Esta implementação da justiça é demasiado forte, em relação ao que verdadeiramente é exigido, mas ganha-se simplicidade no agente que representa a classe.

Exemplifica-se a codificação agora proposta traduzindo a versão recursiva do exemplo do factorial[12]. Neste caso o gestor da classe terá apenas um método (o de nascimento) porque a classe só tem um serviço, que é simultaneamente de nascimento e morte. Logo, é desnecessário ver se existem instâncias ou matá-las. Também por isso não será necessário manter um conjunto com a identidade das instâncias.

Exemplo 4.4.1 *Factorial recursivo*

$[\![\texttt{class}\ \mathrm{Fact} \cdots \texttt{end}\ \mathrm{Fact}]\!] = (g)\ \mathrm{Class}(g)$

$$\texttt{rec}\ \mathrm{Class}\ =\ (\mathit{self})\ \mathit{self} \triangleright [\ \mathit{birth} : (r)\ \nu f\ r \triangleleft \mathit{out} : f, \mathrm{Fact}(f\ [\![1]\!]\ [\![1]\!]), \mathrm{Class}(g)\ ]$$

$$\begin{array}{lll}
\texttt{rec}\ \mathrm{Fact} & = & (\mathit{self}\, a\ c) \\
& & \mathit{self} \triangleright [\ \mathit{calc} : (n\ r)\, \texttt{case}\ a\ \texttt{of} \\
& & \quad 1 :\ \texttt{if}\ [\![n < 2]\!]\ \texttt{then} \\
& & \qquad \nu t\ r \triangleleft \mathit{out} : t, \mathrm{Triv}(t) \\
& & \quad\ \ \texttt{else} \\
& & \qquad \texttt{if}\ [\![c \neq 0]\!]\ \texttt{then} \\
& & \qquad\quad \mathit{self} \triangleleft \mathit{calc} : n\ r, \mathrm{Fact}(\mathit{self}\,[\![2]\!]\ [\![0]\!]) \\
& & \qquad \texttt{else} \\
& & \qquad\quad r \triangleleft \mathit{off} \\
& & \qquad \texttt{fi} \\
& & \quad\ \ \texttt{fi} \\
& & \quad 2 :\ \texttt{if}\ [\![n > 1]\!]\ \texttt{then} \\
& & \qquad \nu s\ g \triangleleft \mathit{birth} : s, \\
& & \qquad\quad s \triangleright [\ \mathit{out} : (i)\ \nu u\ i \triangleleft \mathit{calc} : [\![n-1]\!]\ u, \\
& & \qquad\quad u \triangleright [\ \mathit{out} : (j)\ \nu t\ r \triangleleft \mathit{out} : t, \mathrm{Ntriv}(t\ j\ n)\ \& \\
& & \qquad\qquad \mathit{off} :\ r \triangleleft \mathit{off}\,]] \\
& & \quad\ \ \texttt{else} \\
& & \qquad \texttt{if}\ [\![c \neq 0]\!]\ \texttt{then} \\
& & \qquad\quad \mathit{self} \triangleleft \mathit{calc} : n\ r, \mathrm{Fact}(\mathit{self}\,[\![1]\!]\ [\![0]\!]) \\
& & \qquad \texttt{else} \\
& & \qquad\quad r \triangleleft \mathit{off} \\
& & \qquad \texttt{fi} \\
& & \quad\ \ \texttt{fi} \\
& & \texttt{esac}\,]
\end{array}$$

$$\begin{array}{lll}
\mathrm{Triv} & = & (\mathit{self})\ \mathit{self} \triangleright [\,\mathit{commit} : (r)\ r \triangleleft \mathit{out} : [\![1]\!]\ \& \\
& & \qquad\qquad \mathit{abort} : (r)\ r \triangleleft \mathit{out}\,]
\end{array}$$

[12]Ver Exemplo 2.4.3.

$$\begin{aligned}\text{Ntriv} \;=\; (self\, f\; n)\; self \triangleright [\, & commit : (r)\; \nu c\; f \triangleleft commit : c, \\ & \qquad c \triangleright [\, out : (n!)\; r \triangleleft out : [\![n * n!]\!] \,] \;\& \\ & abort : (r)\; r \triangleleft out \,]\end{aligned}$$

■

Observe-se primeiro o agente Fact que disponibiliza o serviço da classe; Tem três parâmetros: i) um nome para as instâncias; ii) um valor que indica qual a alternativa a testar; iii) um valor que indica o número de vezes que pode ainda testar outras alternativas se a que testou não está activada (de forma a testar todas).

Determinar qual a alternativa a testar primeiro é executar uma alternativa generalizada: quando há um pedido do serviço vê-se primeiro qual a alternativa a executar e depois testa-se a sua activação. Uma alternativa é tratada como um método. Se está activada cria-se um objecto que representa a instância no estado bloqueado, permitindo "commit" e "abort". Se não está, vê-se se se pode ainda testar outras alternativas: se já se "correu" todas responde-se que o serviço não está activado e repõe-se o objecto, instância de Fact, com a indicação da próxima alternativa a testar e que já só pode testar mais n-1 alternativas (sendo n o número total); se não"correu" ainda todas, testa-se a seguinte, decrementando o valor do número de alternativas a testar.

As instâncias do agente Triv representam instâncias da classe Fact no estado bloqueado, por ter sido pedido o serviço *calc* e se ter executado a alternativa *triv* desse serviço. O seu método *commit* devolve o resultado da alternativa: o valor 1. Uma vez que, nem o método *commit*, nem o método *abort* repõem o objecto que representa a instância no estado livre, quando qualquer uma delas é executada a instância "morre".

O agente Ntriv é semelhante ao agente Triv, mas o seu método *commit* invoca o método *commit* de outra instância de Fact, que, ou é outra instância de Triv (e devolve o valor 1, morrendo em seguida), ou é uma instância de Ntriv e, por sua vez, invoca o método *commit* de mais outra instância. Cada instância de Ntriv, quando recebe o resultado da instância que invocou, multiplica-o pelo valor do seu parâmetro e envia o resultado ao seu invocador. Faz-se assim iterativamente o cálculo do factorial.

Para evitar grandes repetições de código e uma vez que o agente que representa a classe não é agora significativamente diferente do anterior, desta vez não se faz a codificação da classe típica[13].

No caso geral, o agente que disponibiliza os serviços é semelhante ao agente Serv da secção anterior, mas tem mais dois parâmetros por cada serviço: um que indica qual a alternativa a testar e outro para se saber quantas mais alternativas se pode testar (antes de se concluir que o serviço não está activado).

Note-se que pode haver vários pedidos "simultâneos" de serviços a determinada classe (encontram-se em paralelo no "ambiente"). Mas não há, nesta implementação, nenhum requisito de justiça na gestão dos pedidos de serviços. Isso será feito na próxima secção, com uma fila que recolhe (e ordena por ordem de chegada[14]) os pedidos de serviços.

---

[13] Pode ver-se no anexo A.

[14] disciplina FIFO: First In First Out

4.4.2 Acções internas

Consideram-se agora classes com animação própria, através de acções internas. Uma instância, a partir do momento em que é criada, começa a executar as suas acções internas (sempre que estiver alguma activada) e pode executar serviços quando solicitada. A semântica dita que sempre que houver acções (internas ou externas) activadas, se escolhe, de forma não determinista, uma para executar. Não pode ser adiada indefinidamente a execução de determinada acção que esteja recorrentemente activada (tem que acabar por ser executada).

A implementação deste requisito de justiça forte na gestão das acções é feita de forma "menos forte" que nas alternativas: basta garantir que uma acção chegará a ser executada se está activada recorrentemente. Para garantir isso faz-se uma escolha aleatória (com distribuição uniforme) entre executar um (eventual) pedido de serviço ou uma das acções. Facilmente se constata que não será nula a probabilidade de que alguma das suas acções que esteja recorrentemente activada seja executada[15].

Esta codificação implica que os agentes sejam mais complexos, como consequência da menor rigidez no critério de justiça.

Os pedidos de serviços vão ser colocados numa fila de espera[16] associada à instância (e capsulada do exterior), que será regularmente consultada, evitando que algum seja indefinidamente protelado[17].

As acções internas não podem estar acessíveis exteriormente à instância. Não se podem colocar como métodos adicionais do objecto que representa a instância e disponibiliza os seus serviços, porque assim não estariam capsulados. Logo, vai ser necessária uma nova codificação da classe, alterando o processo que representa a instância, de forma a separar as acções internas das externas e garantir a animação da instância.

Quer-se que exteriormente à instância só se conheceça a identidade da interface desta; a sua animação deve ser gerida internamente. Como se pode observar no agente apresentado à frente, uma instância será então um processo com animação interna, que cria um conjunto de nomes locais (para denotar os seus serviços e a fila onde são colocados os pedidos de execução desses serviços, por ordem de chegada) e tem um conjunto de objectos, em paralelo, partilhando esses nomes. Esses objectos (instâncias de agentes) são:

- O interface da instância, que disponibiliza os seus serviços (excepto o de nascimento, que está no objecto "gestor" da classe). O seu nome é o identificador dessa instância, que ér conhecido no exterior, uma vez que quando é solicitado o serviço de nascimento é criado um nome para a instância, que é enviado ao invocador desse serviço.

  Este objecto tem que conhecer o nome da fila associada à instância, porque quando recebe um pedido de serviço vai colocar nessa fila o nome do serviço, o valor do parâmetro e o nome onde o invocador do serviço espera a resposta;

[15] Esta implementação da justiça é bastante mais "fiel" à definição do que aquela que foi feita na secção anterior, para as alternativas.

[16] Ver def. 3.1.12.

[17] Esta opção é mais forte que o que é exigido pela semântica, pois os serviços são executados pela ordem de chegada dos pedidos.

- O gestor da animação interna da instância, Anim, que tem um serviço que vai decidir que acção executar (*go*), tendo como parâmetro o atributo da classe.

  Quando recebe uma mensagem de animação faz uma escolha aleatória, com igual probabilidade, entre executar um pedido de serviço ou uma das suas acções internas. Para executar um pedido de serviço, retira o que está há mais tempo à espera da fila (no caso de haver pedidos — se não deve voltar a fazer a escolha de que acção executar) e envia ao gestor das acções externas uma mensagem que solicita esse serviço. Para executar uma acção interna, escolhida aleatoriamente, envia uma mensagem ao gestor das acções internas, lançando o método correspondente essa acção.

  Este objecto tem que conhecer os identificadores do gestor das acções externas e do gestor das acções internas da fila de serviços pedidos e do objecto que gera números aleatórios (instância do agente Random);

- O gestor das acções externas, Ae, que disponibiliza os métodos dos diversos serviços, testando para cada um, rotativamente, as suas alternativas.

  Este objecto tem que conhecer a identidade do gestor da animação interna, o identificador da instância; para cada serviço identifica-se a alternativa a testar e o número de vezes que pode ainda tentar outras alternativas se a que testou não está activada.

  Quando verifica que determinada alternativa do serviço *v* solicitado está activada cria um objecto (instância do agente V) com o método dessa alternativa, que permite executá-la de facto ou desistir da sua execução. Em ambos os casos é depois enviada, a Anim, a ordem para retomar a animação interna (solicitação do seu serviço *go*).

  A este último objecto são também passados o nome do gestor da animação interna da instância, os identificadores dos objectos que disponibilizam os serviços que este método solicita, o parâmetro do serviço a que este objecto corresponde e o valor do seu atributo;

- o gestor das acções internas, Ai, que disponibiliza os métodos das diversas acções internas. Se a acção a executar está activada, faz as diversas chamadas a serviços (de acordo com o seu método), eventualmente altera o atributo enviando o seu novo valor numa mensagem ao gestor da animação interna que dá ordem para passar à proxima acção. Se a acção não está activada é logo enviada a ordem para passar à proxima acção.

  Este objecto tem que conhecer apenas a identidade do gestor da animação interna da instância.

São estes quatro objectos em paralelo, juntamente com a fila de pedidos de serviços, que representam uma instância de uma classe GNOME.

Veja-se com um exemplo como fica codificada uma classe GNOME com acções externas e internas. Trata-se de um objecto que implementa um relógio com minutos e segundos, que tem dois atributos, um serviço com dois resultados e outro com dois parâmetros (excepções ao definido inicialmente, que servem para que a classe tenha duas acções internas, uma que actualiza os minutos e outra os segundos).

Exemplo 4.4.2 *Relógio*

$[\![$`class` Relógio
  `interface`
    `birth serv` *new*
    `serv` *obs*
      `res` *sec:* `nat`
      `res` *min:* `nat`
    `serv` *set*
      `par` *sec:* `nat`
      `par` *min:* `nat`
  `body`
    `slot` *sec:* `nat`
    `slot` *min:* `nat`
    `serv` *new*
      `val` $sec \ll 0$
      `val` $min \ll 0$
    `serv` *obs*
      `ret` $obs.sec = sec$
      `ret` $obs.min = min$
    `serv` *set*
      `val` $sec \ll set.sec$
      `val` $min \ll set.min$
    `act` *tic*
      `enb` $sec < 60$
      `val` $sec \ll sec + 1$
    `act` *min*
      `enb` $sec = 60$
      `val` $sec \ll 0$
      `val` $min \ll min + 1$
`end` Relógio$]\!] = (g)\ \nu l\ \mathrm{Rel}(g\ l), \mathrm{Novo}(l)$

$$\begin{aligned}\mathrm{Rel} \ &= (self\, l)\ \texttt{rec}\ \mathrm{Rel}\\ &\quad self \triangleright [\ birth : (r)\ \nu n\ r \lhd out : n, \mathrm{New}(n\ l), \mathrm{Rel}\ \&\\ &\qquad death : (i\ r)\ l \lhd retirar : i\ ; (r \lhd out, \mathrm{Rel})\ \&\\ &\qquad existe : (i\ r)\ \texttt{if}\ l \lhd existe : i\ \texttt{then}\ r \lhd out, \mathrm{Rel}\\ &\qquad\qquad \texttt{else}\ r \lhd off, \mathrm{Rel}\ \texttt{fi}\ ]\end{aligned}$$

$$\begin{aligned}\mathrm{New} \ &= (self\, l)\ self \triangleright [\, commit : (p\ r)\ \nu i\ l \lhd inserir : i\ ; (r \lhd out : i, \mathrm{InstRel}(i\ [\![0]\!]\ [\![0]\!]))\ \&\\ &\qquad abort : \quad (r)\ r \lhd out\ ]\end{aligned}$$

$$\begin{aligned}\mathrm{InstRel} \ &= (self\ min\ sec)\ \nu\ a\ ae\ ai\ f\ o\\ &\quad \mathrm{Interface}(self\, f), \mathrm{Anim}(a\ ae\ ai\ o\ f), a \lhd go : min\ sec,\\ &\quad \mathrm{Ae}(ae\ a), \mathrm{Ai}(ai\ a), \mathrm{Novafila}(f), \mathrm{Random}(o)\end{aligned}$$

$$\begin{aligned}\mathrm{Interface} \ &= (self\, f)\ \texttt{rec}\ \mathrm{Int}\\ &\quad self \triangleright [\ \ obs : (r)\ f \lhd enqueue : obs\ nil\ r\ ; \mathrm{Int}\ \&\\ &\qquad set : (p_1\ p_2\ r)\ f \lhd enqueue : set\ p_1\ p_2\ r\ ; \mathrm{Int}\ ]\end{aligned}$$

$$
\begin{array}{rcl}
\mathrm{Anim} & = & (\mathit{self}\ ae\ ai\ o\ f)\ \mathtt{rec}\ \mathrm{Anim} \\
& & \quad \mathit{self} \rhd [\ go : (min\ sec)\ \mathtt{case}\ o \lhd val : [\![3]\!]\ of \\
& & \qquad\qquad 1 : \ case\ f \lhd dequeue\ of \\
& & \qquad\qquad\qquad dequeued : (l\ \tilde{p}\ r)\ ae \lhd l : \tilde{p}\ min\ sec\ r \\
& & \qquad\qquad\qquad empty : \mathit{self} \lhd go : min\ sec \\
& & \qquad\qquad\quad \mathtt{esac} \\
& & \qquad\qquad 2 : \ ai \lhd tic : min\ sec \\
& & \qquad\qquad 3 : \ ai \lhd \mathrm{min} : min\ sec \\
& & \qquad\quad \mathtt{esac}, \\
& & \qquad\quad \mathrm{Anim}\ ]
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{Ae} & = & (\mathit{self}\ a)\ \mathtt{rec}\ \mathrm{Ae}\ \mathit{self} \rhd [\ obs : (min\ sec\ r)\ \nu s\ r \lhd out : s, \mathrm{Obs}(s\ a\ min\ sec), \mathrm{Ae}\ \& \\
& & \qquad set : (p_1\ p_2\ min\ sec\ r)\ \nu s\ r \lhd out : s, \mathrm{Set}(s\ a\ p_1\ p_2\ min\ sec), \mathrm{Ae}\ ]
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{Obs} & = & (\mathit{self}\ a\ min\ sec)\ \mathit{self} \rhd [\ commit : (r)\ r \lhd out : min\ sec, a \lhd go : min\ sec\ \& \\
& & \qquad abort : \ (r)\ r \lhd out, a \lhd go : min\ sec\ ]
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{Set} & = & (\mathit{self}\ a\ p_1\ p_2\ min\ sec)\ \mathit{self} \rhd [\ commit : (r)\ r \lhd out, a \lhd go : p_1\ p_2\ \& \\
& & \qquad abort : \ (r)\ r \lhd out, a \lhd go : min\ sec\ ]
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{Ai} & = & (\mathit{self}\ a)\ \mathtt{rec}\ \mathrm{Ai} \\
& & \mathit{self} \rhd [\ tic : (min\ sec)\ \ \mathtt{if}\ [\![sec < 60]\!]\ \mathtt{then}\ a \lhd go : min\ [\![sec + 1]\!] \\
& & \qquad\qquad \mathtt{else}\ a \lhd go : min\ sec\ \mathtt{fi}, \\
& & \qquad\qquad \mathrm{Ai}\ \& \\
& & \quad min : (min\ sec)\ \mathtt{if}\ [\![sec = 60]\!]\ \mathtt{then}\ a \lhd go : [\![min + 1]\!]\ [\![0]\!] \\
& & \qquad\qquad \mathtt{else}\ a \lhd go : min\ sec\ \mathtt{fi}, \\
& & \qquad\qquad \mathrm{Ai}\ ]
\end{array}
$$

■

Quando é criada uma instância da classe (instância do agente InstRel), são lançados um conjunto de objectos — o interface, o gestor da animação interna, o gestor das acções internas e o gestor das acções externas — todos em paralelo, e é enviada uma mensagem para o gestor da animação interna. Esta mensagem desencadeia a animação interna da instância, que fica a alternar entre determinar aleatóriamente a próxima acção a executar (interna ou externa) e a execução dessa acção.

Em paralelo com a animação descrita, a instância pode estar a receber pedidos de serviços, que vai colocando na fila.

Quando é decidido que vai ser executado um serviço, retira-se da fila o pedido que está lá há mais tempo. O identificador do serviço é o primeiro nome da sequência que se retirou da fila. O segundo nome é o eventual parâmetro do serviço e o terceiro o nome de continuação, passado pelo invocador do serviço (e onde este espera o resultado).

No código de Anim, na alternativa *dequeued*, pressupôs-se que o TyCO tem implementada uma operação que transforma nomes em etiquetas[18], de forma a ser pedido (a Ae) o serviço que corresponde ao nome que se retirou da fila.

Para ser executada uma acção interna basta enviar a Ai uma mensagem solicitando o

[18] Explica-se em detalhe no apêndice B.

método correspondente.

Estão agora codificadas em TyCO classes GNOME com as primitivas consideradas no fragmento da linguagem que foi apresentado. Procurou-se que os agentes obtidos para representar as classes permitam obter objectos com o comportamento ditado pela semântica do GNOME. Facilmente se aceita agora que se pode representar qualquer classe GNOME em TyCO, face à tradução definida.

No apêndice A apresenta-se uma classe com estrutura típica e o agente correspondente.

Capítulo 5

Certificação de tipos em TyCO e prova de invariantes

5.1 O sistema de tipos do TyCO

Foi dito na apresentação do cálculo de objectos, no capítulo 3, que os termos eram implicitamente tipificados. De facto, nos termos não há informação explicita sobre os tipos. A tipificação dos termos é feita recorrendo a sistemas que atribuem tipos aos nomes (de processos ou de agentes) e conjuntos de pares nome-tipo (chamados tipificações) aos processos [Vasconcelos 94b, Vasconcelos 94c], que se constituem como *sistemas formais de atribuição de tipos a nomes e tipificações a processos*, semelhantes aos sistemas de tipos do Cálculo-λ [Barendregt 92, Hindley e Seldin 86].

Neste capítulo apresenta-se um desses sistemas, que se verá garantir que a inferência de tipos e de tipificações é computável e que a atribuição de tipos e de tipificações é decidível.

Ver-se-á também que um processo que seja possível tipificar não encontra erros em tempo de execução, fornecendo assim o sistema uma garantia de segurança para os programas TyCO. Além disso, a tipificação de um processo fornece *especificação parcial*, no sentido em que fornece informação sobre o comportamento do processo.

Discute-se ainda a utilização do sistema para provas de correcção parcial de especificações GNOME face a propriedades de invariância.

5.1.1 Tipos e tipificações

Introduzem-se as noções de tipo e tipificação e mais algumas noções auxiliares que serão úteis posteriormente.

Considera-se que o alfabeto do TyCO contém mais um conjunto numerável $\mathbf{V}$, de *variáveis de tipo*, sendo $\mathbf{V}$ disjunto de $\mathbf{L}$.

Definição 5.1.1 O conjunto $\mathbf{T}$ dos *tipos* é definido indutivamente do modo que se segue:

- $t \in \mathbf{T}$ desde que $t \in \mathbf{V}$;
- $[l_1 : \tilde{\alpha}_1, \ldots, l_n : \tilde{\alpha}_n] \in \mathbf{T}$ desde que $l_i \in \mathbf{L}$ e $\tilde{\alpha}_i \in \mathbf{T}^*$, com $l_i \neq l_j$, $\forall i, j : 1 \leq i, j \leq n$;

- $\mu t.\,\alpha \in \mathbf{T}$ desde que $t \in \mathbf{V}$ e $\alpha \in \mathbf{T}$. ■

O registo $[l_1 : \tilde{\alpha}_1, \ldots, l_n : \tilde{\alpha}_n]$ denota um tipo a atribuir a um nome de um objecto com n serviços, etiquetados com $l_1, \ldots, l_n$, cujo método l_n é do tipo $\tilde{\alpha}_i$.

As duas primeiras regras definem os *tipos simples*. Facilmente se observa que processos com estruturas recorrentes de nomes não podem ser tipificados apenas com os tipos simples. Um exemplo simples é uma mensagem que transporte o seu nome alvo. A terceira regra define os *tipos recursivos* e permite tipificar os processos referidos.

Definição 5.1.2 O *tipo recursivo* $\mu t.\alpha$ denota a árvore infinita que é solução da equação $t = \alpha$. Seja α^* essa árvore; se existe uma árvore β^* tal que $\alpha^* = \beta^*$ ter-se-á que o tipo α é *equivalente* ao tipo β, o que se denota por $\alpha \approx \beta$. ■

Definição 5.1.3 A expressão $x : \alpha$ diz-se uma *atribuição de tipo a um nome de processo*, sendo x esse nome, que é chamado o *sujeito* da atribuição e α um tipo, que é chamado o *predicado* da atribuição. ■

Definição 5.1.4 O conjunto de atribuições de tipo $x_1 : \alpha_1, \ldots, x_n : \alpha_n$ diz-se uma *tipificação*, sendo os nomes de processos x_i distintos dois a dois, e α_i tipos. ■

Definição 5.1.5 A expressão $X : \tilde{\alpha}$ diz-se uma *atribuição de tipo a um nome de agente*, sendo X esse nome e $\tilde{\alpha}$ uma sequência de tipos. ■

Definição 5.1.6 O conjunto de atribuições de tipo $X_1 : \tilde{\alpha}_1, \ldots, X_n : \tilde{\alpha}_n$ diz-se uma *base*, sendo X_i nomes de agente, distintos dois a dois, e $\tilde{\alpha}_i$ sequências de tipos. ■

Definição 5.1.7 Duas tipificações, Γ e Δ, dizem-se *compatíveis*, o que se denota por $\Gamma \asymp \Delta$, se $\alpha = \beta$ sempre que $x : \alpha \in \Gamma$ e $x : \beta \in \Delta$. ■

Notação 5.1.8 Consideram-se as seguintes abreviaturas:

- as letras *t,u,...* denotam variáveis de tipo;

  as letras gregas minúsculas denotam tipos de nomes;

  as letras gregas maiúsculas denotam tipificações;

  B,B',... denotam bases;

- $\tilde{x} : \tilde{\alpha}$ denota a tipificação $\{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}$;

- $\Gamma \cdot \tilde{x} : \tilde{\alpha}$ denota $\Gamma \cup \{\tilde{x} : \tilde{\alpha}\}$, desde que os nomes em $\tilde{x}$ não ocorram em Γ;

  $B \cdot X : \tilde{\alpha}$ denota $\Gamma \cup \{X : \tilde{\alpha}\}$, desde que X não ocorra em B;

- $\Gamma_1 \asymp \cdots \asymp \Gamma_n$ denota que as tipificações $\Gamma_1, \ldots, \Gamma_n$ são compatíveis duas a duas. ■

No texto que se segue assumem-se sempre estas abreviaturas.

5.1.2 O sistema de tipos e tipificações

Nesta secção define-se o sistema de atribuição de tipos a nomes e tipificações a processos, que se passa a designar informalmente por sistema de tipos.

Definição 5.1.9 Seja P um termo de processo e A um termo de agente, ambos arbitrários. Seja ainda Γ uma tipificação e $\tilde{\alpha}$ uma sequência arbitrária de tipos;

O conjunto **FT** das *fórmulas* é definido indutivamente do modo que se segue:

- $P \blacktriangleright \Gamma \in \mathbf{FT}$
- $A \blacktriangleright \tilde{\alpha}, \Gamma \in \mathbf{FT}$ ■

A primeira fórmula denota a atribuição de uma tipificação a um processo. A segunda denota a atribuição de tipo a um agente, onde $\tilde{\alpha}$ são os tipos dos nomes que são parâmetros do agente A e Γ é uma tipificação para os nomes livres em A.

Definição 5.1.10 O conjunto $\mathbf{ST}_\mu$, que estabelece o *sistema de tipos recursivo* é definido indutivamente da forma que se segue:

VAR	$B \cdot X : \tilde{\alpha} \vdash X \blacktriangleright \tilde{\alpha}, \emptyset \in \mathbf{ST}_\mu$
$\approx$	$B \vdash P \blacktriangleright \Gamma \cdot x : \beta \in \mathbf{ST}_\mu$ desde que $B \vdash P \blacktriangleright \Gamma \cdot x : \alpha \in \mathbf{ST}_\mu$ e $\alpha \approx \beta$
MSG$_\mu$	$B \vdash a \lhd l_1 : \tilde{v} \blacktriangleright \{a : \beta\} \cup \{\tilde{v} : \tilde{\alpha}_1\} \in \mathbf{ST}_\mu$ desde que $\beta \approx [l_1 : \tilde{\alpha}_1, \ldots, l_n : \tilde{\alpha}_n]$
OBJ	$B \vdash a \rhd [l_1 : A_1 \ \& \ \cdots \ \& \ l_n : A_n] \blacktriangleright \{a : [l_1 : \tilde{\alpha}_1, \ldots, l_n : \tilde{\alpha}_n]\} \cup \Gamma_1 \cup \cdots \cup \Gamma_n \in \mathbf{ST}_\mu$ desde que $B \vdash A_1 \blacktriangleright \tilde{\alpha}_1, \Gamma_1, \ldots, B \vdash A_n \blacktriangleright \tilde{\alpha}_n, \Gamma_n \in \mathbf{ST}_\mu$ e $a : [l_1 : \tilde{\alpha}_1, \ldots, l_n : \tilde{\alpha}_n] \asymp \Gamma_1 \asymp \cdots \asymp \Gamma_n, n \geq 0)$
SCOP	$B \vdash \nu x \ P \blacktriangleright \Gamma \in \mathbf{ST}_\mu$ desde que $B \vdash P \blacktriangleright \Gamma \cdot x : \alpha \in \mathbf{ST}_\mu$
COMP	$B \vdash P, Q \blacktriangleright \Gamma \cup \Delta \in \mathbf{ST}_\mu$ desde que $B \vdash P \blacktriangleright \Gamma \in \mathbf{ST}_\mu$, $B \vdash Q \blacktriangleright \Delta \in \mathbf{ST}_\mu$ e $\Gamma \asymp \Delta$
ABS	$B \vdash (\tilde{x})P \blacktriangleright \tilde{\alpha}, \Gamma \in \mathbf{ST}_\mu$ desde que $B \vdash P \blacktriangleright \Gamma \cdot \tilde{x} : \tilde{\alpha} \in \mathbf{ST}_\mu$
WEAK	$B \vdash P \blacktriangleright \Gamma \cdot x : \alpha \in \mathbf{ST}_\mu$ desde que $B \vdash P \blacktriangleright \Gamma \in \mathbf{ST}_\mu$ e $x \notin \Gamma$
APP	$B \vdash U(\tilde{v}) \blacktriangleright \Gamma \cup \{\tilde{v} : \tilde{\alpha}\} \in \mathbf{ST}_\mu$ desde que $B \vdash U \blacktriangleright \tilde{\alpha}, \Gamma \in \mathbf{ST}_\mu$, com $U \in \mathbf{NA}$ ou $U \in \mathbf{ST}_\mu$ e $\Gamma \asymp \{\tilde{v} : \tilde{\alpha}\}$
LET	$B \vdash \texttt{let } X = A \texttt{ in } P \blacktriangleright \Gamma \cup \Delta \in \mathbf{ST}_\mu$ desde que $B \vdash A \blacktriangleright \tilde{\alpha}, \Gamma \in \mathbf{ST}_\mu$, $B \cdot X : \tilde{\alpha} \vdash P \blacktriangleright \Delta \in \mathbf{ST}_\mu$ e $\Gamma \asymp \Delta$
REC	$B \vdash \texttt{rec } X = A \ \blacktriangleright \tilde{\alpha}, \Gamma \in \mathbf{ST}_\mu$ desde que $B \cdot X : \tilde{\alpha} \vdash A \blacktriangleright \tilde{\alpha}, \Gamma \in \mathbf{ST}_\mu$ ■

Observação 5.1.11 Nomenclatura

- Escreve-se $B \vdash P \blacktriangleright \Gamma$ (respectivamente $B \vdash A \blacktriangleright \tilde{\alpha}, \Gamma$) se a fórmula $P \blacktriangleright \Gamma$ (respectivamente $A \blacktriangleright \tilde{\alpha}, \Gamma$) é *derivável* apartir da base B, usando os axiomas e regras apresentados;
- Seja B uma base e Γ uma tipificação. Se se tem que $B \vdash P \blacktriangleright \Gamma$ então diz-se que P é *tipificável* e que Γ é uma *tipificação bem formada* para P, apartir da base B. ■

O sistema de tipos definido é monomórfico. Mas viu-se já que o cálculo permite definir agentes polimórficos (como, por exemplo, os conjuntos e as filas[1]). Por isso, é necessário estender o conjunto dos tipos introduzindo os tipos polimórficos e estender depois o sistema de tipos.

Definição 5.1.12 O conjunto $\mathbf{T}_{\mu\forall}$ dos *tipos polimórficos* é definido indutivamente do modo que se segue:

- $\alpha \in \mathbf{T}_{\mu\forall}$ desde que $\alpha \in \mathbf{T}$
- $\forall t.\, \sigma \in \mathbf{T}_{\mu\forall}$ desde que $t \in \mathbf{V}$ e $\sigma \in \mathbf{T}_{\mu\forall}$ ∎

Observação 5.1.13 Assume-se que $\sigma[\alpha/t]$ denota a substituição das ocorrências livres da variável de tipo t pelo tipo α. ∎

Definição 5.1.14 O conjunto $\mathbf{ST}_{\mu\forall}$ define o *sistema de tipos polimórfico*; contém os axiomas e regras de $\mathbf{ST}_{\mu}$, substituindo em Let e Var o tipo monomórfico $\tilde{\alpha}$ pelo tipo polimórfico σ, permitindo que nas bases ocorram tipos polimórficos, mas não nas tipificações, e juntando as seguintes regras:

$\forall$-INTRO $B \vdash A \blacktriangleright \forall t.\, \sigma, \Gamma \in \mathbf{ST}_{\mu\forall}$ desde que $B \vdash A \blacktriangleright \sigma, \Gamma \in \mathbf{ST}_{\mu\forall}$ e a variável t é muda em Γ e/ou em B;

$\forall$-ELIM $B \vdash A \blacktriangleright \sigma[\alpha/t], \Gamma \in \mathbf{ST}_{\mu\forall}$ desde que $B \vdash X \blacktriangleright \forall t.\, \sigma, \Gamma \in \mathbf{ST}_{\mu\forall}$ ∎

5.2 Exemplos

Para atribuir uma tipificação a um processo (ou um tipo a um agente) aplicam-se as regras do sistema de tipos aos termos que o constituem, "de dentro para fora", ou seja, começando por tipificar os métodos dos objectos, depois os objectos e finalmente os processos que constituem o processo (ou agente) em causa.

Como se verá, a tipificação de um processo (ou o tipo de um agente) dá informação relevante sobre o seu comportamento admissível. Neste sentido, diz-se que o sistema de tipos fornece *especificação parcial*.

Apresentam-se alguns exemplos de atribuição de tipificações a processos e tipos a agentes, fazendo-se a inferência ao estilo da dedução natural.

Exemplo 5.2.1 *O tipo dos valores booleanos*

Considere-se a primeira codificação proposta para os booleanos, na definição 3.1.5. Aplicando as regras Msg, Abs e Obj obtem-se:

$$\dfrac{\dfrac{\vdash r \lhd \mathit{false} \blacktriangleright \{r : [\mathit{false}{:},\ \mathit{true}{:}]\}}{\vdash (r)\ r \lhd \mathit{false} \blacktriangleright [\mathit{false}{:},\ \mathit{true}{:}]}}{\vdash f \rhd [\mathit{val} : (r)\ r \lhd \mathit{false}] \blacktriangleright \{f : [\mathit{val} : [\mathit{false}{:},\ \mathit{true}{:}]]\}}$$

[1] Ver exemplos 3.1.11 e 3.1.12.

$$\dfrac{\dfrac{\vdash r \lhd true \blacktriangleright \{r : [false:, true:]\}}{\vdash (r)\ r \lhd true \blacktriangleright [false:, true:]}}{\vdash t \rhd [val : (r)\ r \lhd true] \blacktriangleright \{t : [val : [false:, true:]]\}}$$

Pode-se definir a abreviatura de tipo:

$$\texttt{Bool} \stackrel{\text{def}}{=} [val : [false:, true:]]$$

Conclui-se assim que os nomes que denotem estes objectos, que representam os booleanos, têm tipo `Bool`: recebem uma mensagem *val* que pode transportar os nomes *true* ou *false*.
■

Exemplo 5.2.2 *O tipo dos agentes booleanos*
Na definição 3.1.5 foi proposta uma segunda codificação para os booleanos. O tipo do agente False é obtido utilizando as regras Var, App, Comp, Abs, Obj e Rec:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{False} : [val : [false:, true:]] \vdash \text{False} \blacktriangleright [val : [false:, true:]]}{\text{False} : [val : [false:, true:]] \vdash \text{False}(b) \blacktriangleright \{b : [val : [false:, true:]]\}}}{\text{False} : [val : [false:, true:]] \vdash (r)\ r \lhd false, \text{False}(b)] \blacktriangleright [false:, true:], \{b : [val : [false:, true:]]\}}}{\text{False} : [val : [false:, true:]] \vdash b \rhd [val : (r)\ r \lhd false, \text{False}(b)] \blacktriangleright \{b : [val : [false:, true:]]\}}}{\text{False} : [val : [false:, true:]] \vdash (b)\ b \rhd [val : (r)\ r \lhd false, \text{False}(b)] \blacktriangleright [val : [false:, true:]]}}{\vdash \texttt{rec}\ \text{False} = (b)\ b \rhd [val : (r)\ r \lhd false, \text{False}(b)] \blacktriangleright [val : [false:, true:]]}$$

e da mesma forma se infere o tipo do agente True, sendo de novo possível definir a abreviatura `Bool` como o tipo de True e de False. ■

Exemplo 5.2.3 *Uma regra para a alternativa simples*
Definiu-se em 3.1.8 uma alternativa simples como:

$$\texttt{if}\ b\ \texttt{then}\ A_1\ \texttt{else}\ A_2\ \texttt{fi} \stackrel{\text{def}}{=} \nu x\ b \lhd \text{val} : x, x \rhd [true : A_1\ \&\ false : A_2]$$

Seja α o tipo $[true : \tilde{\alpha}_1, false : \tilde{\alpha}_2]$. Pode-se agora deduzir a sua regra de atribuição de tipificação, que se pode juntar ao sistema de tipos definido:

$$\dfrac{\dfrac{B \vdash A_1 \blacktriangleright \tilde{\alpha}_1, \Gamma_1 \quad B \vdash A_2 \blacktriangleright \tilde{\alpha}_2, \Gamma_2}{B \vdash x \rhd [true : A_1\ \&\ false : A_2] \blacktriangleright \{x : \alpha\} \cup \Gamma_1 \cup \Gamma_2} \quad b \lhd \text{val} : x \blacktriangleright \{b : [val : \alpha\}}{B \vdash \texttt{if}\ b\ \texttt{then}\ A_1\ \texttt{else}\ A_2\ \texttt{fi} \blacktriangleright \{b : [val : \alpha\} \cup \Gamma_1 \cup \Gamma_2}$$

A regra deduzida escreve-se da seguinte forma:

If $B \vdash \texttt{if}\ b\ \texttt{then}\ A_1\ \texttt{else}\ A_2\ \texttt{fi} \blacktriangleright \{b : [val : [true : \tilde{\alpha}_1, false : \tilde{\alpha}_2]\} \in \mathbf{ST}_\mu$ desde que
$B \vdash A_1 \blacktriangleright \tilde{\alpha}_1, \Gamma_1 \in \mathbf{ST}_\mu$, $B \vdash A_2 \blacktriangleright \tilde{\alpha}_2, \Gamma_2 \in \mathbf{ST}_\mu$ e $\{x : [true : \tilde{\alpha}_1, false : \tilde{\alpha}_2]\} \asymp \Gamma_1 \asymp \Gamma_2$

■

Exemplo 5.2.4 *Tipo das listas*

Recorde-se que com os agentes Nil e Cons (definidos em 3.1.10) podem-se implementar listas (polimórficas). Ao procurar tipificar o processo $\nu l\ \mathrm{cons}(l\ h\ l), \mathrm{Nil}(l)$, que representa a lista vazia, obtem-se um tipo para as listas:

$$\mathtt{Lista} \stackrel{\mathrm{def}}{=} \forall t.\mu u.[\mathit{val} : [\mathit{nil}:, \ \mathit{cons} : t\ u]]$$

O tipo espelha que o processo que representa uma lista recebe mensagens *val*, respondendo com uma mensagem *nil* se é vazia ou com uma mensagem *cons* que transporta o nome que denota o valor que está à cabeça (que tem um tipo qualquer) e o nome da célula seguinte da lista, se a lista não é vazia. ■

A inferência deste último tipo é já um pouco "pesada". Para processos (ou agentes) mais complexos que os apresentados anteriormente justifica-se a automatização do processo de atribuição de tipificações (e tipos). Existe implementado em SML/NJ, por Vasconcelos, o verificador de tipos `tyco`, com o qual se verificou a correcção (sintática) dos agentes e processos apresentados até agora, e se fez a inferência do seu tipo.

Foi usando o `tyco`, que se obteve o tipo dos seguintes agentes, que se apresentaram no capítulo anterior:

Exemplo 5.2.5 *Tipo do Contador*

O agente Cont do exemplo 4.2.1, que implementa um contador de naturais (tipo `Nat`), tem tipo:

$$[\mathit{inc} :, \mathit{obs} : [\mathit{out} : \mathtt{Nat}], \mathit{set} : \mathtt{Nat}]$$

Como se vê, o contador pode receber pedidos para incrementar o contador (*inc*), para enviar o seu valor corrente (*obs*), ou para inicializar o contador com um valor passado (*set*).

O tipo da versão do contador do exemplo 4.3.1 é bem mais complexo. Justifica-se fazer a inferência através do `tyco`. Para que o exemplo passe no `tyco` tem que se retirar todo o açúcar sintático, expandindo as abreviaturas. Apresentam-se apenas os tipos obtidos.

Considerem-se os tipos:

- $\mathtt{CA} \stackrel{\mathrm{def}}{=} [\mathit{out} : [\mathit{abort} : [\mathit{out} :], \mathit{commit} : [\mathit{out} : \alpha]]]$

  que corresponde aos agentes que implementam o protocolo commit-abort — representam instâncias do contador no estado bloqueado — dependendo o tipo α do que é devolvido pelo método *commit* correspondente ao serviço solicitado;

- $\mathtt{Cont} \stackrel{\mathrm{def}}{=} [\mathit{end} : \mathtt{CA}, \mathit{inc} : \mathtt{CA}, \mathit{obs} : \mathtt{CA}, \mathit{set} : \mathtt{Nat\ CA}]$

  que corresponde ao tipo de uma instância da classe Cont. Em *obs*, o tipo α de `CA` é `Nat`; nos restantes é vazio (a mensagem *out* não transporta nenhum nome).

Relembre-se que o agente Class tem dois parâmetros, um que dará nome ao gestor da classe e outro ao conjunto associado à classe. Vejam-se os tipos desses parâmetros:

- Um tipo do conjunto[2] é:

$$\texttt{Conj} \stackrel{\text{def}}{=} [\mathit{existe} : \texttt{Cont}\ [\mathit{false} :, \mathit{true} :], \mathit{inserir} : \texttt{Cont}\ [\mathit{out} :], \mathit{retirar} : \texttt{Cont}\ [\mathit{out} :]]$$

  que mostra que os métodos do conjunto (*existe*,*inserir*,*retirar*) têm dois parâmetros, sendo o primeiro o elemento a inserir (que neste caso é o nome de uma instância do contador) e o segundo o nome para onde envia a mensagem de sincronização;

- e um tipo para o gestor da classe, instância de Class é:

$$\begin{array}{ll} [\ \mathit{birth} : & [\mathit{out} : [\mathit{abort} : [\mathit{out} :], \mathit{commit} : [\mathit{out} : \texttt{Cont}]]], \\ \ \ \mathit{death} : & \texttt{Cont}\ [\mathit{out} :], \\ \ \ \mathit{existe} : & \texttt{Cont}\ [\mathit{off} :, \mathit{out} :]\ ] \end{array}$$

  que permite verificar que é o método *birth* que disponibiliza um objecto que pode criar instâncias do contador (o seu nome é enviado na mensagem de sincronização de *commit*), e os métodos *death* e *existe* têm dois parâmetros: um do tipo `Cont` e outro que serve para a sincronização.

Constata-se facilmente que o tipo de uma instância do contador não é substancialmente diferente do tipo obtido para o exemplo 4.2.1. Há um padrão comum: ambos são um registo com os nomes dos serviços da classe. ■

Fica claro dos exemplos que a tipificação de um termo fornece informação sobre ele — a sua "interface e os parâmetros de cada método; o tipo dos nomes livres — dizendo-se, por isso, que o sistema de tipos enriquece o TyCO com *especificação parcial*.

5.3 Alguns resultados importantes

Os sistemas de tipos apresentados têm um conjunto de propriedades importantes, das quais se salienta a *invariância do tipo sob a redução*, a *decidibilidade da atribuição de tipos e de tipificações* e a *computabilidade da inferência de tipos e de tipificações*.

Não sendo o objectivo central deste trabalho estudar em detalhe os sistemas de tipos, para a sua melhor compreensão vale a pena juntar alguns dos resultados mais importantes e as suas provas, que surgem em [Vasconcelos 94b, Vasconcelos 94c].

5.3.1 A invariância do tipo sob a redução

Definição 5.3.1 Seja $\Gamma = \{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}$. Define-se:

1. dom $\Gamma = \{x_1, \ldots, x_n\}$;
2. $\Gamma x_i = \alpha_i$;
3. Para $\mathrm{N} \subseteq \mathrm{dom}\,\Gamma$, a *restrição* de Γ a N é a tipificação $\Gamma \restriction \mathrm{N} = \{x : \Gamma x \mid x \in \mathrm{N}\}$;
4. A *redução* de Γ pelo nome x, Γ/x, é a tipificação $\Gamma \restriction (\mathrm{dom}\,\Gamma \backslash \{x\})$. ■

[2] Apresentado no Exemplo 3.1.11.

Igualmente se define dom B, BX_i, $B\restriction \mathrm{N}$ e B/X.

Para as tipificações têm-se os seguintes resultados:

Proposição 5.3.2 : Considere-se $\Gamma \asymp \Delta \asymp \Theta$. Então:

1. $\Gamma \cup \Delta$ é uma tipificação;
2. $\Gamma \cup \Delta \asymp \Theta$;
3. Se $\Gamma' \subseteq \Gamma$ então $\Gamma' \asymp \Delta \asymp \Theta$. ∎

A prova faz-se directamente a partir da definição de compatibilidade de tipificações.

Os sistemas de atribuição de tipificações a processo atribuem tipos aos nomes (de processo ou de agentes) livres desses processos — se um processo é tipificável então existe uma derivação que realiza essa atribuição. O lema seguinte apresenta esta propriedade:

Lema 5.3.3 : Se $B \vdash P \blacktriangleright \Gamma$ então

1. $\mathrm{NL}(P) \subseteq \mathrm{dom}\,\Gamma$ e $B \vdash P \blacktriangleright \Gamma\restriction \mathrm{NL}(P)$
2. $\mathrm{AL}(P) \subseteq \mathrm{dom}\,B$ e $B\restriction \mathrm{AL}(P) \vdash P \blacktriangleright \Gamma$ ∎

Prova: As provas fazem-se por indução na dedução de $B \vdash P \blacktriangleright \Gamma$. Por serem muito semelhantes apresentam-se simultaneamente.
Suponha-se que a dedução é simplesmente o axioma $\textsc{Msg}_\mu$:
$B \vdash a \lhd l_1 : \tilde{v} \blacktriangleright \{a : [l_1 : \tilde{v}_1, \ldots, l_n : \tilde{v}_n], \tilde{v} : \tilde{\alpha}\}$. Então $\mathrm{NL}(P) = \{a, \tilde{v}\} = \mathrm{dom}\,\Gamma$ e $\Gamma\restriction \mathrm{NL}(P) = \Gamma$. Se a dedução termina com:
i) a regra $\approx$ – uma vez que $\mathrm{dom}(\Gamma \cdot x : \alpha) = \mathrm{dom}(\Gamma \cdot x : \beta)$ o resultado verifica-se trivialmente;
ii) a regra $\textsc{Obj}$ – então na dedução surge $B \vdash A_i \blacktriangleright \tilde{\alpha}_i, \Gamma_i$, onde A_i pode ser da forma $(\tilde{x}_i)P_i$ ou $\texttt{rec}\ \mathrm{X}_i = \mathrm{A}_i$; se for o segundo, pela regra $\textsc{Rec}$ surge também na dedução $B \cdot X_i : \tilde{\alpha}_i \vdash A_i \blacktriangleright \tilde{\alpha}_i, \Gamma_i$, $X_i \notin \mathrm{AL}(A_i)$ e por hipótese $\mathrm{AL}(A_i) \subseteq \mathrm{dom}\,B$; para $A_i = (\tilde{x}_i)P_i$ pela regra $\textsc{Abs}$ surge $B \vdash (\tilde{x}_i)P_i \blacktriangleright \tilde{\alpha}_i, \Gamma_i$ com $\tilde{x}_i \notin \mathrm{NL}(P_i)$ e logo, surge também $B \vdash P_i \blacktriangleright \Gamma_i$ sendo $\{a : [l_1 : \tilde{v}_1, \ldots, l_n : \tilde{v}_n]\} \asymp \Gamma_1 \asymp \cdots \asymp \Gamma_n$; começa-se por provar a primeira parte de 1 e 2: pela hipótese de indução $\mathrm{NL}(P_i) \subseteq \mathrm{dom}\,\Gamma_i$ e para qualquer uma das formas de A_i tem-se $\mathrm{NL}(A_i) \subseteq \mathrm{dom}\,\Gamma_i$ e conclui-se então que $\mathrm{NL}(a \rhd [l_1 : A_1 \& \cdots \& l_n : A_n]) \subseteq \mathrm{dom}(\{a : [l_1 : \tilde{v}_1, \ldots, l_n : \tilde{v}_n]\} \cup \Gamma_1 \cup \cdots \cup \Gamma_n)$; de novo pela hipótese de indução $\mathrm{AL}(P_i) \subseteq \mathrm{dom}\,B$ e, pelas regras $\textsc{Rec}$ ou $\textsc{Abs}$, também se tem $\mathrm{AL}(A_i) \subseteq \mathrm{dom}\,B$, vindo $\mathrm{AL}(a \rhd M) \subseteq \mathrm{dom}\,B$. Para provar a segunda parte de 1 veja-se que $B \vdash A_i \blacktriangleright \Gamma_i \restriction \mathrm{NL}(A_i)$ e $\Gamma_i \restriction \mathrm{NL}(A_i) \subseteq \Gamma_i \Longrightarrow (\bigcup \Gamma_i) \restriction (\bigcup \mathrm{NL}(A_i)) \subseteq \bigcup \Gamma_i$. Usando a regra $\textsc{Weak}$ obtem-se $B \vdash A_i \blacktriangleright (\Gamma_i \restriction \mathrm{NL}(A_i)) \cup \{\tilde{x}_i : \tilde{\alpha}_i\} = \Gamma_i$. Por 5.3.2 (3.) vem $\{a : [l_1 : \tilde{v}_1, \ldots, l_n : \tilde{v}_n]\} \asymp (\Gamma_1 \restriction \mathrm{NL}(A_1)) \asymp \cdots \asymp (\Gamma_n \restriction \mathrm{NL}(A_n))$ e aplicando agora a regra $\textsc{Obj}$ deduz-se:
$B \vdash a \rhd [l_1 : A_1 \ \& \cdots \& \ l_n : AP_n] \blacktriangleright$
$\{a : [l_1 : \tilde{v}_1, \ldots, l_n : \tilde{v}_n]\} \cup (\Gamma_1 \restriction \mathrm{NL}(A_1)) \cup \cdots \cup (\Gamma_n \restriction \mathrm{NL}(A_n)) =$
$(\{a : [l_1 : \tilde{v}_1, \ldots, l_n : \tilde{v}_n]\} \cup \Gamma_1 \cup \cdots \cup \Gamma_n) \restriction (\mathrm{NL}(a \rhd [l_1 : A_1 \& \cdots \& l_n : A_n]))$;
Para provar a segunda parte de 2 veja-se que $B \restriction \mathrm{AL}(A_i) \vdash A_i \blacktriangleright \tilde{\alpha}_i, \Gamma_i$ e $\mathrm{AL}(A_i) \subseteq B \Longrightarrow$

$\bigcup(\mathrm{AL}(A_i)) \subseteq B$ vindo $B \upharpoonright \mathrm{AL}(a \triangleright M) \vdash a \triangleright M \blacktriangleright \{a{:}[l_1{:}\tilde{v}_1, \ldots, l_n{:}\tilde{v}_n]\} \cup \Gamma_1 \cup \cdots \cup \Gamma_n$;
iii) a regra SCOP – então $B \vdash P \blacktriangleright \Gamma$; por hipótese de indução $\mathrm{NL}(P) \subseteq \mathrm{dom}\ \Gamma$ e $B \vdash P \blacktriangleright \Gamma \upharpoonright \mathrm{NL}(P)$; usando a regra SCOP deduz-se $B \vdash \nu x P \blacktriangleright (\Gamma \upharpoonright \mathrm{NL}(P))/x$ pois $\mathrm{NL}(\nu x P) = \mathrm{NL}(P) \backslash \{x\} \subseteq \mathrm{dom}\ (\Gamma/x)$, sendo então $(\Gamma \upharpoonright \mathrm{NL}(P))/x = (\Gamma/x) \upharpoonright \mathrm{NL}(\nu x P)$;
iv) a regra WEAK – então na dedução surge $B \vdash P \blacktriangleright \Gamma$ e tome-se um nome x tal que $x \notin \Gamma$; por hipótese de indução $\mathrm{NL}(P) \subseteq \mathrm{dom}\ \Gamma \Longrightarrow x \notin \mathrm{NL}(P)$ e $B \vdash P \blacktriangleright \Gamma \upharpoonright \mathrm{NL}(P)$ deduzindo-se $B \vdash P \blacktriangleright (\Gamma \cup \{x : \alpha\}) \upharpoonright \mathrm{NL}(P)$;
vi) a regra COMP – o resultado sai trivialmente por simples operações de união de conjuntos;
vii) a regra APP – então na dedução surge $B \vdash U \blacktriangleright \tilde{\alpha}, \Gamma$ e $\Gamma \asymp \{\tilde{v} : \tilde{\alpha}\}$; uma vez que $\mathrm{NL(U(\tilde{v})) = NL(U)} \cup \{\tilde{v}\}$ e $\mathrm{AL(U(\tilde{v})) = AL(U)}$, utilizando as hipóteses de indução os resultados saiem trivialmente;
viii) a regra LET – prova-se facilmente pois pela hipótese de indução $\mathrm{NL}(P) \subseteq \mathrm{dom}\ \Delta$ e $\mathrm{NL}(A) \subseteq \mathrm{dom}\ \Gamma$ e ainda porque em `let` $X = A$ `in` P o nome X é mudo. ■

Lema 5.3.4 : Se $B \vdash P \blacktriangleright \Gamma$ e $P \equiv Q$ então $B \vdash Q \blacktriangleright \Gamma$. ■

Prova: A prova faz-se para cada uma das regras que definem a congruência estrutural:
i) para a congruência-alfa prova-se por indução na estrutura de P, assumindo que se $B \vdash P \blacktriangleright \Gamma$ então $B \vdash P[z/x] \blacktriangleright \Gamma[z/x]$, com $x \in \mathrm{NL}(P)$ e $[z/x]$ bem definida;
ii) para as regras 2 a 5 basta mostrar que as árvores de dedução de 2 termos congruentes são equivalentes, o que se faz utilizando a proposição 5.3.2;
iii) para a regra `let` supõe-se que $B \vdash$ `let` $X = A$ `in` $P \blacktriangleright \Gamma$. Pela hipótese de indução tem-se também $B \vdash A \blacktriangleright \tilde{\alpha}, \Delta$ e $B \cdot X : \tilde{\alpha} \vdash P \blacktriangleright \Theta$, com $\Delta \asymp \Theta$ e $\Delta \cup \Theta = \Gamma$. Na dedução de $B \cdot X : \tilde{\alpha} \vdash P \blacktriangleright \Theta$ substitui-se $B \cdot X : \tilde{\alpha} \vdash X(\tilde{v}) \blacktriangleright \Theta \cup \{\tilde{v} : \tilde{\alpha}\}$ por $B \vdash A(\tilde{v}) \blacktriangleright \Delta \cup \{\tilde{v} : \tilde{\alpha}\}$, obtendo-se assim uma dedução de $B \vdash P[X := A] \blacktriangleright \Gamma$; ■

Proposição 5.3.5 : Se $B \vdash P \blacktriangleright \Gamma$ e Q é um subtermo de P então $B \vdash Q \blacktriangleright \Delta$, para uma dada tipificação Δ. ■

A prova faz-se por indução na estrutura de P e é praticamente imediata, considerando-se as hipóteses de indução.

Teorema 5.3.6 (Invariância do tipo sob a redução) Se $B \vdash P \blacktriangleright \Gamma$ e $P \twoheadrightarrow Q$ então $B \vdash Q \blacktriangleright \Gamma$ ■

Prova: O único caso que está por provar é o axioma COMM:
Seja $M = [l_1 : A_1\ \& \cdots \&\ l_n : A_n]$, $C = l_1 : \tilde{v}$ e $\alpha = [l_1 : \tilde{\alpha}_1, \ldots, l_n : \tilde{\alpha}_n]$, e suponha-se que $B \vdash \nu \tilde{x}\ a \triangleright M, a \triangleleft C, \tilde{P} \blacktriangleright \Gamma$. Pela proposição 5.3.5 tem-se igualmente que $B \vdash a \triangleright M, a \triangleleft C \blacktriangleright \Delta$, surgindo então na sua dedução $B \vdash a \triangleright M \blacktriangleright \{a : \alpha\} \cup \Gamma_1 \cup \cdots \cup \Gamma_n$ bem como $B \vdash a \triangleleft C \blacktriangleright \{\tilde{v} : \tilde{\alpha}_1, a : \alpha\}$. Logo $\Delta = \{\tilde{v} : \tilde{\alpha}_1, a : \alpha\} \cup \Gamma_1 \cup \cdots \cup \Gamma_n$ e por APP sai $B \vdash A_1(\tilde{v}) \blacktriangleright \{\tilde{v} : \tilde{\alpha}_1\} \cup \Gamma_1 \subseteq \Delta$; usando a regra WEAK obtem-se $B \vdash A_1(\tilde{v}) \blacktriangleright \Delta \Longleftrightarrow B \vdash M \bullet C \blacktriangleright \Delta$ como se queria provar. ■

O inverso em geral não é verdade — termos não tipificáveis podem reduzir-se para termos tipificáveis ou durante uma redução podem-se perder nomes livres, não coincidindo as tipificações.

Mas há situações em que não é possível realizar uma redução (havendo uma mensagem para determinado nome que identifica um objecto). Se se encarar um processo como um programa, podem acontecer dois tipos de erros em tempo de execução: i) a etiqueta de uma determinada mensagem não identifica nenhum serviço do objecto alvo; ii) o número de nomes na sequência passada por uma mensagem não coincide com os parâmetros do serviço que essa mensagem selecciona.

Define-se em seguida o que é um erro de execução e expõe-se um importante corolário do teorema anterior.

Definição 5.3.7 Um termo de processo P contém um possível *erro em tempo de execução*, o que se denota por $P \in$ Erro, se existe um termo de processo Q tal que $P \twoheadrightarrow Q$, sendo Q da forma $B \vdash \nu\tilde{x}\ a \rhd M, a \lhd C, \tilde{P} \blacktriangleright \Gamma$, mas $M \bullet C$ não está definida, seja porque a etiqueta em C não coincide com nenhuma de M ou o número de elementos na sequência de nomes passados em C não coincide com o número de parâmetros do serviço que a mensagem selecciona. ∎

Corolário 5.3.8 (Ausência de erros em tempo de execução) Se P é tipificável então $P \notin$ Erro. ∎

Prova: Por hipótese P é tipificável e $P \twoheadrightarrow Q$, sendo Q da forma apresentada atrás. Logo, pelo teorema 5.3.6 Q é também tipificável e pela proposição 5.3.5 vem por Msg que $B \vdash a \lhd C \blacktriangleright \{\tilde{v} : \tilde{\alpha}, a : \beta\}$ e por Obj que $B \vdash a \rhd M \blacktriangleright \{a : [l_1 : \tilde{\alpha}_1, \ldots, l_n : \tilde{\alpha}_n]\} \cup \Gamma$; uma vez que, pela regra Comp, se tem agora $B \vdash a \lhd C, a \rhd M \blacktriangleright \Delta$, onde Δ é $\{\tilde{v} : \tilde{\alpha}, a : \beta\} \cup \{a : [l_1 : \tilde{\alpha}_1, \ldots, l_n : \tilde{\alpha}_n]\} \cup \Gamma$ e $\{\tilde{v} : \tilde{\alpha}, a : \beta\} \asymp \{a : [l_1 : \tilde{\alpha}_1, \ldots, l_n : \tilde{\alpha}_n]\} \cup \Gamma$, vem que $\beta = \{a : [l_1 : \tilde{\alpha}_1, \ldots, l_n : \tilde{\alpha}_n]\}$, $\tilde{\alpha} = \tilde{\alpha}_i$, $1 \leq i \leq n$ e $\mathrm{length}(\tilde{v}) = \mathrm{length}(\tilde{x}_i)$. ∎

5.3.2 Duas propriedades

Os dois teoremas seguintes são fundamentais para a automatização dos sistemas de atribuição de tipificações a termos de processo. No entanto, a sua demonstração exige que se verifique a propriedade da existência de uma tipificação mais geral, o que não se verifica nos sistemas apresentados [Vasconcelos 94b, Vasconcelos 94c]. Só fazendo uma extensão destes sistemas, introduzindo *géneros* é possível verificar a propriedade e demonstrar os teoremas. Mas isso ultrapassa já o âmbito desta dissertação.

Teorema 5.3.9 (Decidibilidade da atribuição de tipos e de tipificações)

1. Dados um termo de processo P, uma base B e uma tipificação Γ pode-se determinar se $B \vdash P \blacktriangleright \Gamma$.

2. Dados um termo de agente A, uma base B e um tipo α pode-se determinar se $B \vdash A \blacktriangleright \alpha$. ∎

Teorema 5.3.10 (Computabilidade da inferência de tipos e de tipificações)

1. Dado um termo de processo P pode-se determinar se existe uma tipificação Γ tal que $B \vdash P \blacktriangleright \Gamma$. Além disso, essa tipificação pode ser calculada a partir de P.

2. Dado um termo de agente A e uma base B pode-se determinar se existe um tipo α tal que $B \vdash A \blacktriangleright \alpha$. Além disso, esse tipo pode ser calculado a partir de A e de B. ∎

5.4 Cálculo de invariantes

Como se viu, o sistema de atribuição de tipificações a processos TyCO permite garantir a ausência de erros em tempo de execução. Esta importante propriedade, junto com o facto de o sistema atribuir tipo aos nomes, permite pensar que é possível usar o sistema para provar a *correcção parcial de especificações* GNOME, *face a propriedades de segurança* (ou de invariância), desde que estas se reduzam a saber se uma variável em determinado programa (neste caso, um nome num processo) pertence a um dado conjunto (neste caso, o tipo) que expressa a propriedade. Esta conjectura baseia-se no facto de a invariância do tipo durante a redução garantir que um nome mantém o seu tipo durante a execução do programa, e, uma vez que o sistema de tipos permite inferir o tipo do nome em questão, pode-se verificar se esse tipo satisfaz a propriedade que se procura provar.

5.4.1 Estratégia da prova

Segundo [Ben-Ari 90], se $\mathrm{P}(\tilde{x})$ for uma propriedade das variáveis de entrada $\tilde{x}$ e $\mathrm{Q}(\tilde{x},\tilde{y})$ for uma propriedade das variáveis de entrada $\tilde{x}$ e de saída $\tilde{y}$, para um conjunto de valores $\tilde{a}$ das variáveis de entrada e um conjunto de valores $\tilde{b}$ das variáveis de saída, define-se a *correcção parcial* como:

$$(\mathrm{P}(\tilde{a}) \wedge \mathrm{Termina}(\mathrm{Prog}(\tilde{a},\tilde{b}))) \Longrightarrow \mathrm{Q}(\tilde{a},\tilde{b})$$

Uma *propriedade de segurança* é, neste caso, uma propriedade dos atributos que deve ser sempre verdadeira ao longo da vida dessa classe. Em [Manna e Pnueli 92] são agrupadas em duas famílias, de acordo com o tipo de garantia que fornecem: exclusão mútua e ausência de impasse.

Assumindo que estas propriedades podem ser traduzidas em afirmações do tipo "*um atributo só toma valores num dado intervalo*", sendo esse intervalo recursivamente enumerável, seguindo as razões explicadas atrás, defende-se que se pode recorrer ao sistema de tipos para as provar.

Chama-se a essas afirmações *invariantes* sobre especificações GNOME e vai-se procurar usar o sistema de tipos para provar esses invariantes (considerando apenas invariantes locais a classes).

Fixa-se a seguinte estratégia de prova: i) determina-se o intervalo que traduz o invariante; ii) na especifição GNOME, altera-se o tipo do atributo, passando a ser o intervalo obtido; iii) gera-se um novo processo TyCO para representar a classe GNOME, com nova codificação para o tipo de dados do atributo, que passou a ser esse intervalo; iv) faz-se a inferência do tipo desse agente que representa a classe e analisa-se o tipo do atributo em questão: se for o intervalo (ou um seu subconjunto) o invariante é respeitado; caso contrário não o é.

5.4.2 Uma prova e uma contra-prova

Nesta secção final exemplifica-se a estratégia de prova proposta, fazendo-se uma prova de uma propriedade de invariancia sobre uma especificação, e uma contra-prova. discute-se finalmente o exemplo e as provas, expondo as limitações do sistema de tipos para o cálculo de invariantes.

Considere-se a classe Sem, definida no exemplo 4.2.2. Suponha-se que sobre essa especificação se pretende provar o seguinte invariante: *o atributo só pode assumir os valores 0 e 1*. Logo, de acordo com a estratégia de prova fixada, o primeiro passo é alterar, na especificação, o tipo do atributo em causa: na declaração de s o seu tipo passa a ser o tipo enumerado [0,1], subtipo dos naturais. É necessário agora codificar este tipo em TyCO e inferir o tipo do novo agente que representa a classe, resultado da alteração no tipo de dados do atributo, verificando então se o invariante é ou não respeitado por esse agente.

O agente TyCO que corresponde à classe Sem é complexo, devido à necessidade de permitir que seja pedido um serviço a uma instância da classe mas depois não se o execute (o que é implementado com o protocolo "commit-abort"). Para o efeito aqui pretendido esse processo é mesmo demasiado complexo, sendo a atribuição de tipo fastidiosa. Por isso vai-se retirar do agente o protocolo referido, conjecturando que isso não altera significativamente o tipo do nome que denota o atributo. Esta conjectura é suportada pelos testes feitos no `tyco`, de que se apresentou atrás um caso, no exemplo 5.2.5.

A classe será representada por um agente (que não obedece completamente à semântica do GNOME, não tendo a estrutura proposta no final do capítulo 4 porque lhe foi retirado o protocolo "commit-abort"), assumindo-se estarem disponíveis, no ambiente, os processos que implementam o tipo de dados considerado para o atributo (neste caso, [0,1]).

Uma instância de Sem é criada pelo processo seguinte:

$$
\begin{array}{l}
\texttt{let rec}\ \mathrm{Sem} = (i\ s)\\
\qquad\qquad\quad i \triangleright [\, \mathit{req} : \nu b\ \ \mathit{gtz} \triangleleft \mathrm{val} : s\ b,\\
\qquad\qquad\qquad\qquad\qquad b \triangleright [\, \mathit{true} : \nu r\ \ \mathit{pred} \triangleleft \mathrm{val} : s\ r,\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad r \triangleright [\ \mathrm{val} : (v)\ \mathrm{Sem}(i\ v)\ ]\ \&\\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{false} : \mathrm{Sem}(i\ s)\ ]\ \&\\
\qquad\qquad\qquad\quad \mathit{rel} : \nu b\ \ \mathit{lto} \triangleleft \mathrm{val} : s\ b,\\
\qquad\qquad\qquad\qquad\qquad b \triangleright [\, \mathit{true} : \nu r\ \ \mathit{succ} \triangleleft \mathrm{val} : s\ r,\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad r \triangleright [\ \mathrm{val} : (v)\ \mathrm{Sem}(i\ v)\ ]\ \&\\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{false} : \mathrm{Sem}(i\ s)\ ]]\\
\texttt{in}\ \mathrm{Sem}(i\ \mathit{one})
\end{array}
$$

que está em paralelo com o processo que representa o intervalo natural [0,1]:

$$\texttt{rec}\ X.\mathit{zero} \triangleright [\ \mathrm{val} : (r)\ r \triangleleft \mathit{zero}, X\ ],$$

$$\texttt{rec}\ X.\mathit{one} \triangleright [\ \mathrm{val} : (r)\ r \triangleleft \mathrm{one}, X\ ],$$

$$
\begin{array}{l}
\texttt{rec}\ X.\mathit{prd} \triangleright [\ \mathrm{val} : (s\ r)\ \nu x\ \ s \triangleleft \mathrm{val} : x,\\
\qquad\qquad\qquad\qquad\qquad x \triangleright [\ \mathit{zero} : r \triangleleft \mathrm{val} : \mathit{zero}\ \&\ \mathit{one} : r \triangleleft \mathrm{val} : \mathit{zero}\ ]], X,
\end{array}
$$

$$
\begin{array}{l}
\texttt{rec}\ X.\mathit{suc} \triangleright [\ \mathrm{val} : (s\ r)\ \nu x\ \ s \triangleleft \mathrm{val} : x,\\
\qquad\qquad\qquad\qquad\qquad x \triangleright [\ \mathit{zero} : r \triangleleft \mathrm{val} : \mathit{one}\ \&\ \mathit{one} : r \triangleleft \mathrm{val} : \mathit{one}\ ]], X,
\end{array}
$$

$$\texttt{rec}\ X.\mathit{lto} \triangleright [\ \mathrm{val} : (s\ r)\ \nu x\ s \triangleleft \mathrm{val} : x, x \triangleright [\ \mathit{zero} : r \triangleleft \mathit{true}\ \&\ \mathit{one} : r \triangleleft \mathit{false}\ ]], X,$$

$$\texttt{rec}\ X.\mathit{gtz} \triangleright [\ \mathrm{val} : (s\ r)\ \nu x\ s \triangleleft \mathrm{val} : x, x \triangleright [\ \mathit{zero} : r \triangleleft \mathit{false}\ \&\ \mathit{one} : r \triangleleft \mathit{true}\ ]], X$$

Seguidamente faz-se a inferência do tipo de Sem. Considerem-se os seguintes tipos:

- $\sigma = [\mathit{rel}:, \mathit{req}:]$;
- $[0,1] = [\mathit{val}: [\mathit{one}:, \mathit{zero}:]]$;
- $\alpha = [\mathit{val}: [0,1]]$;
- $\beta = [\mathit{val}: [0,1]\ \alpha]$, que corresponde a um tipo funcional $[0,1] \to [0,1]$;
- $\texttt{Bool} = [\mathit{false}:, \mathit{true}:]$;
- $\gamma = [\mathit{val}: [0,1]\ \texttt{Bool}]$, que corresponde a um tipo funcional $[0,1] \to \texttt{Bool}$.

Facilmente se verifica que os objectos *zero* e *one* têm tipo [0,1], os objectos *suc* e *prd* têm tipo β e os objectos *lto* e *gtz* têm tipo γ.

Vai-se agora procurar atribuir uma tipificação ao agente Sem. Começa-se pelo serviço *req*; seja:

$$\mathrm{Req} = \nu b\ gtz \triangleleft \mathit{val}: s\ b, b \triangleright [\ \mathit{true}: \mathrm{Pred}(i\ s)\ \&\ \mathit{false}: \mathrm{Sem}(i\ s)\ ]$$

$$\mathrm{Pred} = (i\ s)\ \nu r\ prd \triangleleft \mathit{val}: s\ r, r \triangleright [\ \mathit{val}: (v)\ \mathrm{Sem}(i\ v)\ ]$$

Naturalmente que o tipo do nome de uma instância de Sem é σ e o tipo do seu atributo é [0,1]. Deduz-se então, aplicando as regras Var, App, Obj, Obj, Comp e Abs,:

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathrm{Sem}:\sigma[0,1] \vdash \mathrm{Sem} \blacktriangleright \sigma[0,1]}{\mathrm{Sem}:\sigma[0,1] \vdash \mathrm{Sem}(i\ s) \blacktriangleright \{i:\sigma, s:[0,1]\}}}{\mathrm{Sem}:\sigma[0,1] \vdash r \triangleright [\mathit{val}:(v)\ \mathrm{Sem}(i\ v)] \blacktriangleright \{r:\alpha, i:\sigma\}} \quad \vdash prd \triangleleft \mathit{val}: s\ r \blacktriangleright \{prd:\beta, s:[0,1], r:\alpha\}}{\mathrm{Sem}:\sigma[0,1] \vdash prd \triangleleft \mathit{val}: s\ r, r \triangleright [\ \mathit{val}: (v)\ \mathrm{Sem}(i\ v)\ ] \blacktriangleright \{prd:\beta, s:[0,1], r:\alpha, i:\sigma\}}}{\mathrm{Sem}:\sigma[0,1] \vdash \mathrm{Pred}(i\ s) \blacktriangleright \{prd:\beta, i:\sigma, s:[0,1]\}}$$

Seja agora $\mathrm{Alt} = b \triangleright [\ \mathit{true}: \mathrm{Pred}(i\ s)\ \&\ \mathit{false}: \mathrm{Sem}(i\ s)\ ]$. A dedução segue com as regras Obj, Msg, Comp e Scop:

$$\dfrac{\dfrac{\mathrm{Sem}:\sigma[0,1] \vdash \mathrm{Alt} \blacktriangleright \{b:\texttt{Bool}, prd:\beta, i:\sigma, s:[0,1]\} \quad \vdash gtz \triangleleft \mathit{val}: s\ b \blacktriangleright \{b:\texttt{Bool}, gtz:\gamma, s:[0,1]\}}{\mathrm{Sem}:\sigma[0,1] \vdash gtz \triangleleft \mathit{val}: s\ b, \mathrm{Alt} \blacktriangleright \{b:\texttt{Bool}, gtz:\gamma, prd:\beta, i:\sigma, s:[0,1]\}}}{\mathrm{Sem}:\sigma[0,1] \vdash \mathrm{Req} \blacktriangleright \{gtz:\gamma, prd:\beta, i:\sigma, s:[0,1]\}}$$

de onde se conclui que, no serviço *req*, o atributo s só pode assumir os valores 0 e 1, verificando-se o invariante.

Para o serviço *rel* o procedimento é idêntico. Seja:

$$\mathrm{Rel} = \nu b\ lto \triangleleft \mathit{val}: s\ b, b \triangleright [\ \mathit{true}: \mathrm{Succ}(i\ s)\ \&\ \mathit{false}: \mathrm{Sem}(i\ s)\ ]$$

$$\mathrm{Succ} = (i\ s)\ \nu r\ suc \triangleleft \mathit{val}: s\ r, r \triangleright [\ \mathit{val}: (v)\ \mathrm{Sem}(i\ v)\ ]$$

$$\text{Alt} = b \triangleright [\ \mathit{true} : \text{Succ}(i\ s)\ \&\ \mathit{false} : \text{Sem}(i\ s)\ ]$$

Deduz-se agora $\text{Sem}\!:\!\sigma[0,1] \vdash \text{Rel} \blacktriangleright \{lto\!:\!\gamma, suc\!:\!\beta, i\!:\!\sigma, s\!:\![0,1]\}$. Conclui-se também que, no serviço *rel*, o atributo s só pode assumir os valores 0 e 1, verificando-se igualmente o invariante. Vem então, pela regra Obj, tendo como hipóteses as duas deduções anteriores:

$$\text{Sem}\!:\!\sigma[0,1] \vdash \text{Sem}(i\ s) \blacktriangleright \{gtz\!:\!\gamma, prd\!:\!\beta, lto\!:\!\gamma, suc\!:\!\beta, i\!:\!\sigma, s\!:\![0,1]\}$$

e logo, o agente Sem respeita o invariante pois, na tipificação do processo que representa uma sua instância, o atributo s só pode assumir os valores 0 e 1.

Falta apenas ver se o invariante é verificado no momento inicial (no nascimento da instância): a execução do serviço de nascimento lança o processo $\text{Sem}(i\ \mathit{one})$, onde o nome i da instância tem tipo σ e o nome *one* tem o tipo [0,1]. Pela regra App obtem-se a tipificação $\{i : \sigma, \mathit{one} : [0,1]\}$ para $\text{Sem}(i\ \mathit{one})$ e verifica-se que também o serviço de nascimento respeita o invariante.

Conclui-se finalmente que o invariante se verifica em toda a vida de qualquer instância da classe, pois é estabelecido com o nascimento dessa instância e cada uma das suas acções o preserva.

Importa igualmente ver se se pode verificar que um invariante não é verificado. Suponha-se então que a condição de activação do serviço *rel* era `enb` $s < 2$ e que se pretende verificar o mesmo invariante. Esta alteração na condição de activação exige nova codificação do tipo de dados do atributo — o processo TyCO que representa o tipo de dados passa a ser:

$$
\begin{aligned}
&\texttt{rec}\ X.\mathit{ltt} \triangleright [\ \mathit{val} : (s\ r)\ \nu x\ \ s \lhd \mathit{val} : x,\\
&\qquad\qquad x \triangleright [\mathit{zero} : r \lhd \mathit{true}, X\ \&\\
&\qquad\qquad\quad \mathit{one} : r \lhd \mathit{true}, X\ \&\\
&\qquad\qquad\quad \mathit{two} : r \lhd \mathit{false}, X\ ]],\\
&\texttt{rec}\ X.\mathit{gtz} \triangleright [\ \mathit{val} : (s\ r)\ \nu x\ \ s \lhd \mathit{val} : x,\\
&\qquad\qquad x \triangleright [\mathit{zero} : r \lhd \mathit{false}, X\ \&\\
&\qquad\qquad\quad \mathit{one} : r \lhd \mathit{true}, X\ \&\\
&\qquad\qquad\quad \mathit{two} : r \lhd \mathit{true}, X\ ]],\\
&\texttt{rec}\ X.\mathit{suc} \triangleright [\ \mathit{val} : (s\ r)\ \nu x\ \ s \lhd \mathit{val} : x,\\
&\qquad\qquad x \triangleright [\mathit{zero} : r \lhd \mathit{val}\!:\!\mathit{one}, X\ \&\\
&\qquad\qquad\quad \mathit{one} : r \lhd \mathit{val}\!:\!\mathit{two}, X\ \&\\
&\qquad\qquad\quad \mathit{two} : r \lhd \mathit{val}\!:\!\mathit{two}, X\ ]],\\
&\texttt{rec}\ X.\mathit{prd} \triangleright [\ \mathit{val} : (s\ r)\ \nu x\ \ s \lhd \mathit{val} : x,\\
&\qquad\qquad x \triangleright [\mathit{zero} : r \lhd \mathit{val}\!:\!\mathit{zero}, X\ \&\\
&\qquad\qquad\quad \mathit{one} : r \lhd \mathit{val}\!:\!\mathit{zero}, X\ \&\\
&\qquad\qquad\quad \mathit{two} : r \lhd \mathit{val}\!:\!\mathit{one}, X\ ]],\\
&\texttt{rec}\ X.\mathit{zero} \triangleright [\ \mathit{val} : (r)\ r \lhd \mathit{zero}, X\ ],\\
&\texttt{rec}\ X.\mathit{one} \triangleright [\ \mathit{val} : (r)\ r \lhd \mathit{one}, X\ ],\\
&\texttt{rec}\ X.\mathit{two} \triangleright [\ \mathit{val} : (r)\ r \lhd \mathit{two}, X\ ]
\end{aligned}
$$

Além destas alterações, no agente Sem, no método do serviço *rel*, em vez de se ter uma mensagem *val* para *lto* passa-se a ter essa mensagem mas para *ltt*. De resto o agente fica inalterado.

Ao procurar tipificar Sem verifica-se que os objectos *zero*, *one* e *two* passam a ter tipo [0,1,2] (que é abreviatura de [*val*:[*one*:,*two*:,*zero*:]]), os objectos *suc* e *prd* passam a ter tipo [*val*:[0,1,2] [*val*:[0,1,2]]] e os objectos *ltt* e *gtz* passam a ter tipo [*val*:[0,1,2] `Bool`], o que leva a que se deduza a seguinte tipificação para uma instância do agente Sem:

$$\mathrm{Sem}:\sigma[0,1] \vdash \mathrm{Sem}(i\ s) \blacktriangleright \{i:\sigma, s:[0,1,2]\}$$

Conclui-se assim que o atributo s pode assumir os valores 0, 1 e 2, não se verificando o invariante nesta versão da classe Sem.

Mas há situações em que não é possível mostrar, através do sistema de tipos, que um invariante não é verificado por determinada especificação. Sempre que a especificação alterada (com o novo tipo para o atributo, tipo esse que expressa o invariante) puder acarretar erros em tempo de execução no GNOME (o atributo pode assumir valores que não pertencem a esse intervalo, violando o invariante), a impossibilidade de ocorrerem erros em tempo de execução em TyCO (por 5.3.8) fará com que se dê como provado um invariante que, na verdade, não se verifica. Se se reparar na codificação das operações sobre o intervalo, constata-se que essas operações têm que ser implementadas em TyCO como funções totais, resultando que um invariante que não se verifica numa especificação GNOME é verificado o agente TyCO correspondente.

Para melhor compreensão do problema assuma-se que, na classe Sem, se retira a condição de activação do serviço *req*. O parâmetro passa a puder ser incrementado indefinidamente, podendo tomar qualquer valor natural. Logo, o invariante não se verifica. Na especificação alterada puder-se-á incorrer num erro em tempo de execução (instanciar o atributo com o valor 2). Mas isto nunca acontecerá no processo TyCO. Uma vez que se codificou que o sucessor de 1 é 1 e que o predecessor de 0 é 0 (só assim — ou de forma semelhante — o sistema de tipos aceita o processo que representa o tipo de dados, pois as funções têm que ser totais), o atributo pertence sempre ao intervalo [0,1] e o invariante não é violado (como deveria ser).

Conclui-se por isso que o sistema de tipos não é adequado a provas de invariância sob especificações GNOME. O problema reside na codificação do tipo de dados do atributo. Mas não parece haver solução para tal, pois qualquer operação desse tipo de dados tem sempre que devolver valores pertencentes ao intervalo que representa o tipo. O carácter estático do sistema de tipos, que não detecta se determinado método de um objecto é ou não alguma vez solicitado (detecta apenas que é possível invoca-lo), impede as provas de invariância.

Capítulo 6

Conclusões

Este capítulo procura resumir o trabalho apresentado e os resultados obtidos, apontar perspectivas de trabalho futuro, que se abrem a partir do estudo aqui feito, e nomear áreas de investigação relacionadas com as questões tratadas.

6.1 Observações finais

Apresenta-se um fragmento da linguagem GNOME, de especificação formal de sistemas concorrentes, reactivos e interactivos, através da sua gramática e da descrição informal da sua semântica. A linguagem é orientada a objectos e está baseada em classes, sendo os objectos instâncias das classes. Uma especificação define um comunidade de objectos, para a qual se pretende definir uma semântica operacional.

Apresenta-se também um Cálculo de Objectos Concorrentes, o TyCO, implicitamente tipificado, na linha do Cálculo-π e do Cálculo-ν, com comunicação assíncrona e passagem de nomes, como nas Álgebras de Processos. O mecanismo de computação é baseado num esquema de redução — a comunicação entre objectos, por emissão de mensagens — que se define.

Faz-se a definição de novos construtos TyCO, que serão úteis na codificação das primitivas do GNOME em TyCO, mostra-se como codificar tipos de dados e como implementar comunicação síncrona a partir do mecanismo de comunicação assíncrona do TyCO.

Define-se depois a semântica operacional do GNOME em TyCO, considerando que uma classe deve ser representada por um agente e, em consequência, um objecto, instância de uma classe, é representado por um processo que é a composição concorrente de um conjunto de instâncias de agentes. Defende-se que o comportamento deste processo é equivalente ao de um objecto, inserido na comunidade de objectos que a especificação define.

Na definição do mapa semântico tratam-se vários aspectos da semântica do GNOME:

- a atomicidade da execução de uma acção, que é implementada com comunicação síncrona entre todos os objectos envolvidos nessa acção e seguindo um protocolo do tipo "commit-abort";
- a concorrência entre objectos animados, tendo cada um comportamento sequencial,

mas realizando, em paralelo, a recepção de pedidos de serviços e a execução de acções;

- o não determinismo interno de um objecto, na escolha da acção a executar, que se implementa, com uma escolha aleatória (com distribuição uniforme) — no caso das acções — ou rotativamente, testando todas, no caso das alternativas;
- a criação e morte dinâmicas dos objectos (que são persistentes, por omissão).

As opções tomadas na codificação das classes resultam de um processo "construtivo", que partiu das primitivas mais simples e foi juntando as mais exigentes, tendo-se procurado obter um agente final o mais simples possível. Em cada passo as opções foram justificadas, tendo sempre como objectivo a simplicidade e a clareza. No entanto, admite-se que existem outras opções alternativas, que por ventura permitem chegar a "melhores" resultados, pois em muitos pontos as opções são múltiplas.

O mapa semântico torna claros os aspectos mais elaborados (e de mais exigente tradução) do GNOME e os seus aspectos mais simples.

Apresenta-se finalmente um sistema de tipos do TyCO, que atribui tipos aos nomes e tipificações aos processos. Para exemplificar a sua aplicação tipificam-se alguns dos termos apresentados no capítulo 3 e dois dos agentes propostos no capítulo 4, salientando-se a capacidade de especificação parcial dos tipos, que constituem um verdadeiro protocolo de comunicações para os processos. Mostram-se algumas das suas propriedades, tais como a invariância do tipo sob a redução, a ausência de erros em tempo de execução nos processos tipificáveis, a computabilidade da inferência de tipos e de tipificações e a decidibilidade da sua atribuição.

Discute-se por fim como usar o sistema de tipos para fazer provas de correcção parcial das especificações face certas a propriedades de invariância, concluindo-se que tal não é possível. Devido à codificação dos tipos de dados, onde as operações devem ser totais, de forma a não haver erros em tempo de execução (única forma aceite pelo sistema de tipos), constatou-se que há situações em que o código GNOME acarreta possíveis erros em tempo de execução, não sendo isso admissível nos processos TyCO bem tipificados. Nessas situações, um invariante que é violado pela especificação GNOME é verificado pelo agente TyCO, contrariando as pretensões de usar o sistema de tipos para provas de invariância.

6.2 Trabalho futuro

Ficam em aberto um conjunto de aspectos que abrem perspectivas para mais trabalho de investigação. Nomeadamente, importa estudar:

- a semântica em TyCO de um fragmento mais alargado da linguagem GNOME, que inclua a composição de especificações, os mecanismos de herança e as primitivas para trabalho cooperativo, como as regiões;
- a automação da tradução das classes em agentes;
- mecanismos formais de verificação da equivalência dos comportamentos das classes GNOME e dos agentes TyCO que os representam;

- o desenvolvimento de um sistema de tipos semelhante aos do TyCO, mas directamente para o GNOME, de forma a garantir a ausência de erros em tempo de execução.

6.3 Outras linhas de investigação

Existem outras semânticas para o GNOME, além da proposta e das já referidas, nomeadamente baseadas em:

- álgebras de objectos [Aiguier *et al* 95];
- estruturas de eventos [Caleiro 95];
- quantales (estudo em curso);
- autómatos não-sequênciais [Menezes *et al* 95];
- cálculo de situações (estudo em curso);

Estão ainda em curso trabalhos que visam estender a linguagem para suportar especificações orientadas a características ("Feature-Oriented"), sendo uma característica *um aspecto ou parte de uma especificação que o utilizador vê e que têm uma função própria* [Guerra e Ryan 96].

Esta é uma nova área de desenvolvimento das especificações formais, por solicitação da indústria, nomeadamente das companhias de telefones.

Outras linhas de investigação em curso visam obter extensões da OSL para lógicas que suportem raciocínio temporal com tempo real [Gomes e Sernadas 95], que suportem raciocínio abdutivo e raciocínio não monótono [Gouveia e Sernadas 93].

Bibliografia

[Agha 86]
G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* M.I.T. Press, 1986.

[Aiguier *et al* 95]
M. Aiguier, G. Bernot, J. Ramos, e A. Sernadas. An algebraic semantics for GNOME via a translation to ETOILE specifications. Relatório interno, Universidade d'Evry, França, 1995. Submetido para publicação.

[Barendregt 74]
H. P. Barendregt. The lambda-calculus, its syntax and semantics. Em *Studies in Logic and the Foundation of Mathematics*, volume 103. North-Holland, 1974. Edição revista.

[Barendregt 92]
H. P. Barendregt. Lambda-calculi with types. Em D. M. Gabbay S. Abramsky e T. S. E. Maibaum, editores, *Handbook of Logic in Computer Science*, volume 2, pp 118–309. Oxford University Press, 1992.

[Ben-Ari 90]
M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* C. A. R. Hoare Series Editor – Prentice-Hall Int., 1990.

[Caleiro 95]
C. Caleiro. Comunidades distribuídas de objectos. Dissertação de Mestrado, IST, Universidade Técnica de Lisboa, 1995. Orientada por A. Sernadas.

[Carmo e Penedo 94]
P. Carmo e P. Penedo. O compilador GNOME. Relatório interno, Secção de Ciência da Computação, Departamento de Matemática, Instituto Superior Técnico, 1096 Lisboa, Portugal, 1994.

[ESDI 93]
ESDI. OBLOG-CASE V1.0. Manual do utilizador, Av. Alvares Cabral, 41, 1200 Lisboa, 1993.

[Gomes e Sernadas 95]
J. Gomes e C. Sernadas. Propositional temporal logic with real time. Relatório interno, Secção de Ciência da Computação, Departamento de Matemática, Instituto Superior Técnico, 1096 Lisboa, Portugal, 1995.

[Gouveia e Sernadas 93]
P. Gouveia e C. Sernadas. Abduction in temporal logic specification. Relatório interno, Secção de Ciência da Computação, Departamento de Matemática, Instituto Superior Técnico, 1096 Lisboa, Portugal, 1993.

[Guerra e Ryan 96]
S. Guerra e M. Ryan. Towards feature-oriented specifications. Em P.-Y. Schobbens, editor, *ModelAge'96: Proceedings of the Esprit WG Workshop*, 1996.

[Hewitt *et al* 73]
C. Hewitt, P. Bishop, e R. Steiger. A universal, modular actor formalism for artificial intelligence. Em *3th International Joint Conference on Artificial Intelligence*, pp 235–245, 1973.

[Hindley e Seldin 86]
J. R. Hindley e J. P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, 1986.

[Hoare 85]
C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Honda e Tokoro 91]
K. Honda e M. Tokoro. An object calculus for asynchronous communication. Em *5th European Conference on Object-Oriented Programming*, volume 512 de LNCS, pp 141–162. Springer-Verlag, 1991.

[Jungclaus *et al* 91]
R. Jungclaus, G. Saake, T. Hartmann, e C. Sernadas. *Object-oriented specification of information systems: The TROLL language*. Universidade Técnica de Braunschweig, Alemanha, 1991.

[LaLonde e Pugh 90]
W. R. LaLonde e J. R. Pugh. *Inside Smalltalk*, volume 1. Prentice-Hall International, 1990.

[Manna e Pnueli 92]
Z. Manna e A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[Menezes *et al* 95]
P. B. Menezes, A. Sernadas, e J. F. Costa. Nonsequential automata semantics for a concurrent object-based language. Relatório interno, Secção de Ciência da Computação, Departamento de Matemática, Instituto Superior Técnico, 1096 Lisboa, Portugal, 1995. Submetido para publicação.

[Milner 89]
R. Milner. *Communication and Concurrency*. C. A. R. Hoare Series Editor – Prentice-Hall Int., 1989.

[Milner 91]
R. Milner. The polyadic π-calculus: a tutorial. Relatório interno ECS-LFCS 91-180, Universidade de Edinburgo, Reino Unido, 1991.

[Milner *et al* 92]
R. Milner, J. Parrow, e D. Walker. A calculus of mobile processes, i e ii. *Information and Computation*, 100:1–77, 1992. Também disponível como Relatório Interno ECS-LFCS 89-85/86, Universidade de Edinburgo.

[Nierstrasz 89]
O. Nierstrasz. Process semantics of temporal specifications. Em W. Kim e F. Lochovski, editores, *Object Oriented Concepts*, pp 3–22. ACM Press, 1989.

[Ramos 94]
J. Ramos. A lógica de certificação GNOME. Dissertação de Mestrado, IST, Universidade Técnica de Lisboa, 1994. Orientada por A. Sernadas.

[Ramos e Sernadas 95a]
J. Ramos e A. Sernadas. A brief introduction to GNOME. Relatório interno, Secção de Ciência da Computação, Departamento de Matemática, Instituto Superior Técnico, 1096 Lisboa, Portugal, 1995.

[Ramos e Sernadas 95b]
J. Ramos e A. Sernadas. GNOME certification logic. Relatório interno, Secção de Ciência da Computação, Departamento de Matemática, Instituto Superior Técnico, 1096 Lisboa, Portugal, 1995.

[Sernadas *et al* 92a]
A. Sernadas, J. F. Costa, e C. Sernadas. Especificação de objectos com diagramas: Abordagem OBLOG. Relatório interno, Secção de Ciência da Computação, Departamento de Matemática, Instituto Superior Técnico, 1096 Lisboa, Portugal, 1992. Prémio Descartes do Ministério das Finanças.

[Sernadas *et al* 92b]
C. Sernadas, P. Gouveia, e A. Sernadas. OBLOG: Object-oriented, logic-based conceptual modeling. Relatório interno, Secção de Ciência da Computação, Departamento de Matemática, Instituto Superior Técnico, 1096 Lisboa, Portugal, 1992.

[Sernadas *et al* 95]
A. Sernadas, C. Sernadas, e J. F. Costa. Object specification logic. *Journal of Logic and Computation*, 5(5):603–630, 1995. Disponível como Relatório Interno do DMIST desde 1992.

[Sernadas e Ramos 94]
A. Sernadas e J. Ramos. A linguagem GNOME: Sintaxe, semântica e cálculo. Relatório interno, Secção de Ciência da Computação, Departamento de Matemática, Instituto Superior Técnico, 1096 Lisboa, Portugal, 1994.

[Sernadas e Sernadas 94]
A. Sernadas e C. Sernadas. Object certification. Relatório interno, Secção de Ciência da Computação, Departamento de Matemática, Instituto Superior Técnico, 1096 Lisboa, Portugal, 1994. Apresentado no DAISD Workshop 94 - Submetido para publicação.

[Tel 94]
G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[Vasconcelos 94a]
V. Vasconcelos. An operacional semantics and a typing system for ABCL/1 based on a calculus of objects. Relatório interno CS 94-001, Universidade de Keio, Japão, 1994.

[Vasconcelos 94b]
V. Vasconcelos. Predicative polymorphism in π-calculus. Em *6th Parallel Architectures and Languages Europe*, volume 817 de LNCS, pp 425–437. Springer-Verlag, 1994.

[Vasconcelos 94c]
V. Vasconcelos. *A Process-Calculus Approach to Typed Concurrent Objects*. Dissertação de Doutoramento, Departamento de Ciência da Computação, Universidade de Keio, Japão, 1994.

[Vasconcelos 94d]
V. Vasconcelos. Typed concurrent objects. Em *8th European Conference on Object-Oriented Programming*, volume 821 de LNCS, pp 100–117. Springer-Verlag, 1994.

[Wegner 90]
P. Wegner. Concepts and paradigms of object-oriented programming. *OOPS MESSENGER*, 1(1):7–87, 1990.

[yon90]
Em A. Yonezawa, editor, *ABCL, an Object-Oriented Concurrent System*. MIT Press, 1990.

Apêndice A

Codificação de uma classe GNOME típica em TyCO

A.1 Semântica da classe GNOME típica em TyCO

Apresenta-se aqui, em detalhe, o agente TyCO que se pretende que tenha um comportamento equivalente a uma classe Gnome com uma estrutura típica. O código dessa classe Gnome com estrutura típica é depois apresentado na segunda secção.

Assume-se que essa classe típica tem as restrições já introduzidas no capítulo 4: cada classe tem apenas um atributo; cada serviço tem, no máximo, um parâmetro e um resultado; cada método (das acções internas ou externas) tem, no máximo, uma condição de activação, uma valoração e um retorno de valores (no caso dos serviços); não há alternativas nas acções internas.

Para maior simplicidade do código TyCO que se apresenta coloca-se apenas o corpo de um dos serviços e de uma das acções — para cada um é semelhante — e considera-se que cada acção não faz mais que duas chamadas de serviços.

Seja então C uma classe Gnome com estrutura típica; o termo TyCO que tem um comportamento equivalente é um agente, denotado por C, que tem como parâmetros:

- o nome que denotará o gestor da classe, instância de C (considera-se apenas um destes elementos por classe);
- a indicação de qual é a primeira alternativa a testar do serviço de nascimento;
- a indicação do número de vezes que pode ainda tentar outras alternativas desse serviço se a que testou não estava activada (de forma a testar todas, sendo $b.n$ o número total de alternativas);
- a identidade do conjunto das instâncias da classe existentes.

Em C apresenta-se apenas o código da primeira alternativa do serviço de nascimento. As restantes são semelhantes.

Veja-se então o agente que representa a classe:

$$[\![\texttt{class}\ \mathrm{C} \cdots \texttt{end}\ \mathrm{C}]\!] = (g)\, \nu l\ \mathrm{C}(g\ 1\ b.n - 1\ l), \mathrm{Novo}(l)$$

onde Novo(l) é um processo que representa o conjunto das instâncias da classe existentes, e C é o agente:

$$
\begin{array}{l}
\texttt{rec}\ \mathrm{C}\ \texttt{=}\ (\mathit{self}\ a\ c\ l) \\
\quad \mathit{self} \triangleright [\,\mathit{birth} : (p\ r)\ \texttt{case}\ a\ \texttt{of} \\
\qquad\qquad 1 :\ \texttt{if}\ [\![\mathrm{act}_{b.1}]\!]\ \texttt{then} \\
\qquad\qquad\quad \texttt{if}\ \mathit{self} \triangleleft \mathit{existe} : [\![\mathrm{alvo}^1_{b.1}]\!]_{a_1}\ \texttt{then} \\
\qquad\qquad\qquad a_1 \triangleleft \mathrm{v}_1 : [\![\mathrm{arg}^1_{b.1}]\!]_{g_1}\ ; \\
\qquad\qquad\qquad (\texttt{if}\ \mathit{self} \triangleleft \mathit{existe} : [\![\mathrm{alvo}^2_{b.1}]\!]_{a_2}\ \texttt{then} \\
\qquad\qquad\qquad\quad a_2 \triangleleft \mathrm{v}_2 : [\![\mathrm{arg}^2_{b.1}]\!]_{g_2}\ ; (\mathrm{Commit} \mid \mathrm{Abort}) \\
\qquad\qquad\qquad\ \ \texttt{else} \\
\qquad\qquad\qquad\quad \mathrm{Abort} \\
\qquad\qquad\qquad\ \ \texttt{fi} \\
\qquad\qquad\qquad\ \ \mid \mathrm{Next}\ ) \\
\qquad\qquad\quad \texttt{else} \\
\qquad\qquad\qquad \mathrm{Next} \\
\qquad\qquad\quad \texttt{fi} \\
\qquad\qquad\ \ \texttt{else} \\
\qquad\qquad\quad \mathrm{Next} \\
\qquad\qquad\ \ \texttt{fi} \\
\qquad\qquad \cdots \\
\qquad\quad \texttt{esac}\ \& \\
\quad \mathit{death} : (i\ r)\, l \triangleleft \mathit{retirar} : i\ ; (r \triangleleft \mathit{out}, \mathrm{C}(\mathit{self}\ a\ c\ l))\ \& \\
\quad \mathit{existe} : (i\ r)\, \texttt{if}\ l \triangleleft \mathit{existe} : i\ \texttt{then}\ r \triangleleft \mathit{out}, \mathrm{C}(\mathit{self}\ a\ c\ l) \\
\qquad\quad \texttt{else}\ r \triangleleft \mathit{off}, \mathrm{C}(\mathit{self}\ a\ c\ l)\ \texttt{fi}\ ]
\end{array}
$$

$\mathrm{Commit} = \nu n\ r \triangleleft \mathit{out} : n, \mathrm{New}_1(n\ \mathit{self}\ v_1\ v_2\ g_1\ g_2\ p\ l), \mathrm{C}(\mathit{self}\ 2\ c\ l)$

$\mathrm{Abort} = a_1 \triangleleft \mathit{abort}\ ; \mathrm{Next}$

Next é o processo que vai testar as alternativas seguintes (se existirem):

$$
\begin{array}{l}
\mathrm{Next} = \texttt{if}\ c \neq 0\ \texttt{then} \\
\qquad \mathit{self} \triangleleft \mathit{birth} : p\ r, \mathrm{C}(\mathit{self}\ 2\ c - 1\ l) \\
\quad\ \ \texttt{else} \\
\qquad r \triangleleft \mathit{off}, \mathrm{C}(\mathit{self}\ a\ c\ l) \\
\quad\ \ \texttt{fi}
\end{array}
$$

Se uma das alternativas testadas está activada (seja, por exemplo, a i) é criado um objecto (instância de New_i) que permite criar a instância da classe, executando as restantes cláusulas dessa alternativa do serviço de nascimento (método *commit*), ou desistir da criação (método *abort*). Se não está nenhuma alternativa activada informa-se o invocador do serviço disso mesmo (pela mensagem *off*).

O agente New_1 corresponde a ter sido tomada a primeira alternativa do serviço de nascimento. O parâmetro g identifica o gestor da classe; v_1 e v_2 são nomes passados pelos serviços invocados e denotam objectos que disponibilizam as instâncias das classes onde ocorrem esses serviços, no estado bloqueado; g_1 e g_2 são nomes que denotam os valores dos argumentos das chamadas; p é o parâmetro do serviço (pode intervir nas expressões das cláusulas); l é o conjunto das instâncias da classe, onde vai ser inserido o identificador da nova instância criada.

$$
\begin{aligned}
\mathrm{New}_1 = (&\mathit{self}\ g\ v_1\ v_2\ g_1\ g_2\ p\ l) \\
&\mathit{self} \triangleright [\ \ \mathit{commit} : (r)\ \nu i\ l \triangleleft \mathit{inserir} : i\ ; (v_1 \triangleleft \mathit{commit} : g_1\ ; v_2 \triangleleft \mathit{commit} : g_2\ ; \\
&\qquad\qquad\qquad\qquad (r \triangleleft \mathit{out} : [\![\mathrm{ret}_{b.1}]\!]\ i, \mathrm{InstC}(i\ g\ [\![\mathrm{val}_{b.1}]\!]))) \ \& \\
&\qquad\quad \mathit{abort} : (r)\ v_1 \triangleleft \mathit{abort}\ ; v_2 \triangleleft \mathit{abort}\ ; r \triangleleft \mathit{out}\ ]
\end{aligned}
$$

Se o serviço de nascimento é executado cria-se uma instância da classe. As instâncias de InstC representam as instâncias da classe. Disponibilizam a interface e têm animação interna. O nome da instância denota também o objecto que representa a sua interface. Uma instância tem que conhecer o identificador da classe e o nome do atributo (são-lhe passados como parâmetros).

$$
\begin{aligned}
\mathrm{InstC} = (&\mathit{self}\ g\ \mathit{slot})\ \nu\ v_1\ \cdots\ v_n\ d\ a\ ae\ ai\ f\ o \\
&\mathrm{Interface}(\mathit{self}\ ae\ v_1\ \cdots\ v_n\ d\ f), \mathrm{Anim}(a\ ae\ ai\ f\ o), a \triangleleft go : \mathit{slot}, \mathrm{Random}(o), \\
&\mathrm{Ae}(ae\ a\ \mathit{self}\ g\ 1\ n.v_1 - 1\ \cdots\ 1\ n.v_n - 1\ 1\ n.d - 1), \mathrm{Ai}(ai\ g\ a), \mathrm{Novafila}(f)
\end{aligned}
$$

O interface da instância disponibiliza os seus serviços (excepto o de nascimento, que está no objecto "gestor" da classe). Tem como parâmetros os nomes que denotam os serviços da classe (nomes locais a InstC) e o nome da fila onde são colocados os pedidos de execução desses serviços, por ordem de chegada. Tem ainda um serviço (*kill*) que, quando a instância é morta, esvazia a fila dos pedidos de execução dos serviços, responde aos invocadores que não está activada e "mata" no final o objecto que representa o interface da instância. Esse serviço recebe um nome como parâmetro que deve ser o do gestor das acções externas: serve para verificar que a ordem não está a ser dada de fora da instância (uma vez que este serviço não está capsulado); se for esse o caso repõe o objecto Interface.

$$
\begin{aligned}
\mathrm{Interface} = (&\mathit{self}\ ae\ v_1\ \cdots\ v_n\ d\ f)\ \mathtt{rec}\ \mathrm{Int} \\
&\mathit{self} \triangleright [\ \mathrm{v}_1 : (p\ r)\ f \triangleleft \mathit{enqueue} : v_1\ p\ r\ ; \mathrm{Int}\ \& \\
&\qquad \cdots \\
&\qquad \mathit{kill} : (o)\ \mathtt{if}\ o = ae\ \mathtt{then}\ f \triangleleft \mathit{clean}\ \mathtt{else}\ \mathrm{Int}\ \mathtt{fi}\ ]
\end{aligned}
$$

O gestor da animação interna da instância, Anim tem um serviço que decide que acção executar (*go*), tendo como parâmetro o atributo da classe, e outro serviço que "mata" o objecto, no caso de ter sido invocado o seu serviço de morte. Nos seus parâmetros estão os identificadores do gestor das acções externas e do gestor das acções internas (a quem se pedem as execuções de acções), da fila de serviços pedidos e do objecto que gera números aleatórios (instância do agente Random).

$$
\begin{array}{l}
\texttt{rec}\ \mathrm{Anim} = (\mathit{self}\ ae\ ai\ f\ o) \\
\quad \mathit{self} \triangleright [\ go : (s)\ \texttt{case}\ o \triangleleft val : m+1\ \texttt{of} \\
\qquad\qquad 1:\ \texttt{case}\ f \triangleleft dequeue\ \texttt{of} \\
\qquad\qquad\qquad dequeued : (l\ p\ r)\ ae \triangleleft l : p\ s\ r \\
\qquad\qquad\qquad empty : \mathit{self} \triangleleft go : s \\
\qquad\qquad\quad \texttt{esac} \\
\qquad\qquad 2:\ ai \triangleleft \mathrm{a}_1 : s \\
\qquad\qquad \ldots \\
\qquad\qquad m+1:\ ai \triangleleft \mathrm{a}_m : s \\
\qquad\quad \texttt{esac}, \mathrm{Anim}(\mathit{self}\, ae\ ai\ f\ o)\ \& \\
\quad\quad \mathit{kill} :\ \mathbf{0}\ ]
\end{array}
$$

O gestor das acções externas, Ae, disponibiliza os métodos dos diversos serviços, testando para cada um, rotativamente, as suas alternativas, e tem também um serviço de morte. Nos seus parâmetros estão a identidade do gestor da animação interna, o identificador da instância, para cada serviço identifica-se a alternativa a testar e o número de vezes que pode ainda tentar outras alternativas se a que testou não está activada.

Se o serviço solicitado é de morte, nos parâmetros passa-se, além dos argumentos usuais para os outros serviços, ainda as identidades da instância e dos gestores da classe e da animação interna e externa. Se for confirmada a execução deste serviço são enviadas ordens de execução dos serviços de morte a esses objectos, de forma a retirá-los do ambiente, e também a ordem para retirar o identificador da instância da lista de instâncias da classe.

$$
\begin{array}{l}
\texttt{rec}\ \mathrm{Ae} = (\mathit{self}\, a\ i\ g\ a_1\ n_1\ \cdots\ a_n\ n_n\ a_d\ n_d) \\
\quad \mathit{self} \triangleright [\ \mathrm{v}_1 : (p\ s\ r)\ \texttt{case}\ a_1\ \texttt{of} \\
\qquad\qquad 1:\ \texttt{if}\ [\![\mathrm{act}_{v.1}]\!]\ \texttt{then} \\
\qquad\qquad\quad \texttt{if}\ \mathit{self} \triangleleft \mathit{existe} : [\![\mathrm{alvo}^1_{v_1.1}]\!]_{a_1}\ \texttt{then} \\
\qquad\qquad\qquad a_1 \triangleleft \mathrm{v}_1 : [\![\mathrm{arg}^1_{v_1.1}]\!]_{g_1}\ ; \\
\qquad\qquad\qquad (\texttt{if}\ \mathit{self} \triangleleft \mathit{existe} : [\![\mathrm{alvo}^2_{v_1.1}]\!]_{a_2}\ \texttt{then} \\
\qquad\qquad\qquad\qquad a_2 \triangleleft \mathrm{v}_2 : [\![\mathrm{arg}^2_{v_1.1}]\!]_{g_2}\ ; (\mathrm{Commit} \mid \mathrm{Abort}) \\
\qquad\qquad\qquad\quad \texttt{else} \\
\qquad\qquad\qquad\qquad \mathrm{Abort} \\
\qquad\qquad\qquad\quad \texttt{fi} \\
\qquad\qquad\qquad \mid \mathrm{Next}) \\
\qquad\qquad\quad \texttt{else} \\
\qquad\qquad\qquad \mathrm{Next} \\
\qquad\qquad\quad \texttt{fi} \\
\qquad\qquad \texttt{else} \\
\qquad\qquad\quad \mathrm{Next} \\
\qquad\qquad \texttt{fi} \\
\qquad\quad \texttt{esac}\ \& \\
\quad\quad \ldots \\
\quad\quad \mathit{kill} :\ \mathbf{0}\ ]
\end{array}
$$

$$\mathrm{Commit} = \nu v\ r \triangleleft out : v, \mathrm{V}(v\ a\ v_1\ v_2\ g_1\ g_2\ p\ s), \mathrm{Ae}(\mathit{self}\, a\ i\ g\ 2\ n_1 - 1\ \cdots\ a_n\ n_n\ a_d\ n_d)$$

$$\mathrm{Abort} = a_1 \triangleleft abort\ ; \mathrm{Next}$$

$$
\begin{array}{l}
\mathrm{Next} = \texttt{if}\ n_1 \neq 0\ \texttt{then} \\
\qquad \mathit{self} \lhd \mathrm{v}_1 : s, \mathrm{Ae}(\mathit{self}\ a\ i\ g\ 2\ n_1 - 1\ \ldots\ a_n\ n_n\ a_d\ n_d\ p\ s\ r) \\
\quad\ \texttt{else} \\
\qquad r \lhd \mathit{off}, a \lhd \mathit{go} : s, \mathrm{Ae}(\mathit{self}\ a\ i\ g\ a_1\ n_1\ \ldots\ a_n\ n_n\ a_d\ n_d\ p\ s\ r) \\
\quad\ \texttt{fi}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{V} = (\mathit{self}\ a\ v_1\ v_2\ g_1\ g_2\ p\ s) \\
\quad \mathit{self} \rhd [\ \mathit{commit} : (r)\ v_1 \lhd \mathit{commit} : g_1\ ; v_2 \lhd \mathit{commit} : a_2\ ; \\
\qquad\qquad\qquad\qquad (r \lhd \mathit{out} : [\![\mathrm{ret}_{v.1}]\!], a \lhd \mathit{go} : [\![\mathrm{val}_{v.1}]\!])\ \& \\
\qquad\quad \mathit{abort} : (r)\ v_1 \lhd \mathit{abort}\ ; v_2 \lhd \mathit{abort}\ ; (r \lhd \mathit{out}, a \lhd \mathit{go} : s)\ ]
\end{array}
$$

$$
\begin{array}{l}
\mathrm{D} = (\mathit{self}\ a\ ae\ ai\ v_1\ v_2\ g_1\ g_2\ p\ s\ c\ g) \\
\quad \mathit{self} \rhd [\ \mathit{commit} : (r)\ v_1 \lhd \mathit{commit} : g_1\ ; v_2 \lhd \mathit{commit} : g_2\ ; \\
\qquad\qquad\qquad\qquad (c \lhd \mathit{kill} : ae, a \lhd \mathit{kill}, ae \lhd \mathit{kill}, ai \lhd \mathit{kill}, g \lhd \mathit{death} : c\ ; \\
\qquad\qquad\qquad\qquad\ (r \lhd \mathit{out} : [\![\mathrm{ret}_{d.1}]\!])))\ \& \\
\qquad\quad \mathit{abort} : (r)\ v_1 \lhd \mathit{abort}\ ; v_2 \lhd \mathit{abort}\ ; (r \lhd \mathit{out}, a \lhd \mathit{go} : s)\ ]
\end{array}
$$

O gestor das acções internas, Ai, disponibiliza os métodos das diversas acções internas, e tem um serviço de morte. Se a acção a executar está activada, faz as diversas chamadas a serviços de outras classes (de acordo com o seu método), eventualmente altera o atributo enviando o seu novo valor numa mensagem ao gestor da animação interna que dá ordem para passar à proxima acção. Se a acção não está activada é logo enviada a ordem para passar à proxima acção. Nos seus parâmetros está a identidade do gestor da animação interna da instância.

$$
\begin{array}{l}
\texttt{rec}\ \mathrm{Ai} = (\mathit{self}\ g\ a) \\
\qquad \mathit{self} \rhd [\ \mathrm{a}_1 : (s)\ \texttt{if}\ [\![\mathrm{act}_{a_1}]\!]\ \texttt{then} \\
\qquad\qquad\qquad\qquad \texttt{if}\ g \lhd \mathit{existe} : [\![\mathrm{alvo}^1_a]\!]_{a_1}\ \texttt{then} \\
\qquad\qquad\qquad\qquad\quad a_1 \lhd \mathrm{v}_1 : [\![\mathrm{arg}^1_{v_1.1}]\!]_{g_1}\ ; \\
\qquad\qquad\qquad\qquad\quad (\texttt{if}\ g \lhd \mathit{existe} : [\![\mathrm{alvo}^2_a]\!]_{a_2}\ \texttt{then} \\
\qquad\qquad\qquad\qquad\qquad a_2 \lhd \mathrm{v}_2 : [\![\mathrm{arg}^2_{v_1.1}]\!]_{g_2}\ ; (\mathrm{Commit} \mid \mathrm{Abort}) \\
\qquad\qquad\qquad\qquad\quad\ \texttt{else} \\
\qquad\qquad\qquad\qquad\qquad \mathrm{Abort} \\
\qquad\qquad\qquad\qquad\quad\ \texttt{fi} \\
\qquad\qquad\qquad\qquad\quad\ \mid \mathrm{Go}) \\
\qquad\qquad\qquad\qquad \texttt{else} \\
\qquad\qquad\qquad\qquad\quad \mathrm{Go} \\
\qquad\qquad\qquad\qquad \texttt{fi} \\
\qquad\qquad\qquad\quad \texttt{else} \\
\qquad\qquad\qquad\qquad \mathrm{Go} \\
\qquad\qquad\qquad\quad \texttt{fi}\ \& \\
\qquad\qquad \ldots \\
\qquad\qquad \mathit{kill} :\ \mathbf{0}\ ]
\end{array}
$$

$$\mathrm{Commit} = v_1 \lhd \mathit{commit} : g_1\ ; v_2 \lhd \mathit{commit} : g_2\ ; a \lhd \mathit{go} : [\![\mathrm{val}^2_a]\!], \mathrm{Ai}(\mathit{self}\ g\ a)$$

$$\mathrm{Abort} = v_1 \lhd \mathit{abort}\ ; \mathrm{Go}$$

$$\mathrm{Go} = a \lhd \mathit{go} : s, \mathrm{Ai}(\mathit{self}\ g\ a)$$

A.2 Estrutura típica de uma classe em GNOME

class C
 interface
  birth serv b
   par $par : gp_{c.b}$
   res $res : gr_{c.b}$
  serv v_1
   par $par : gp_{c.v_1}$
   res $res : gr_{c.v_1}$
  ...
  serv v_n
   par $par : gp_{c.v_n}$
   res $res : gr_{c.v_n}$
  death serv d
   par $par : gp_{c.d}$
   res $res : gr_{c.d}$
 body
  slot $slot : g_c$
  serv b
   ...
  serv v
   alt n
    enb $\mathrm{act}_{v.n}$
    call v_1 of $\mathrm{alvo}^1_{v.n} : c_1$
     arg $v_1.par = \mathrm{arg}^1_{v.n}$
    ...
    call v_k of $\mathrm{alvo}^k_{v.n} : c_k$
     arg $v_k.par = \mathrm{arg}^k_{v.n}$
    val $slot \ll \mathrm{val}_{v.n}$
    ret $v.res = \mathrm{ret}_{v.n}$
   ...
  ...
  serv d
   ...
  act a
   enb act_a
   call v_1 of $\mathrm{alvo}^1_a : c_1$
    arg $v_1.par = \mathrm{arg}^1_a$
   ...
   call v_k of $\mathrm{alvo}^k_a : c_k$
    arg $v_k.par = \mathrm{arg}^k_a$
   val $slot \ll \mathrm{val}_a$
  ...
end C ■

Apêndice B

Algum “açúcar sintáctico” usado no TyCO

Os termos de processo definidos no Capítulo 3 estabelecem uma linguagem que, por vezes, se torna pesada de usar, com construções que se repetem amiúde. Por isso, ao longo do trabalho, são introduzidos, como abreviaturas, construtores auxiliares. Este apêndice resume esses construtores TyCO, que se utilizam, mas que não estão disponíveis no verificador de tipos `tyco` que está implementado.

Definiram-se seis construtores que estendem a linguagem TyCO:

1. as duas alternativas simples, definidas em 3.1.8;
2. as duas alternativas generalizadas, definidas em 3.1.9;
3. a sincronização simples e a sincronização com alternativa, definidas em 3.1.13;
4. a transformação de etiquetas em nomes:

   Pressupõe-se que em TyCO, uma vez seleccionado um serviço de um objecto, a etiqueta desse serviço pode ser usada no seu método como um nome, como por exemplo no processo seguinte:

   $$c \triangleright [l : (\tilde{x})\ a \triangleleft e : \tilde{y}\ l\ \tilde{z}, \tilde{P}\ \&\cdots]$$

5. a transformação de nomes em etiquetas:

   Pressupõe-se que em TyCO, um nome passado como parâmetro de um serviço, pode ser usado como etiqueta no seu método, como por exemplo no processo seguinte:

   $$c \triangleright [e : (\tilde{x} l \tilde{y})\ a \triangleleft l : \tilde{v}, \tilde{P}\ \&\cdots]$$

Para verificar no `tyco` os exemplos apresentados ao longo desta dissertação é necessário expandir as abreviaturas referidas em 1, 2 e 3. Para implementar um compilador da linguagem Gnome em TyCO já será indispensável que as potencialidades descritas em 5 e 6 sejam incorporadas no `tyco`, o que provavelmente acontecerá na sua próxima versão.