UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

UMA LINGUAGEM DE PROGRAMAÇÃO COM TIPOS DE SESSÃO INDEPENDENTES DO CONTEXTO

Bernardo Pinto de Almeida

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

Dissertação orientada por:
Prof. Doutor Vasco Manuel Thudichum de Serpa Vasconcelos

2018

Agradecimentos

Agradeço ao meu orientador, Vasco Vasconcelos pela oportunidade de fazer um projeto numa área tão interessante. Agradeço a sua constante disponibilidade e apoio durante o desenvolvimento deste trabalho de modo a torná-lo o melhor possível. Muito obrigado professor, foi um prazer trabalhar consigo.

Gostaria de agradecer à FCUL que permitiu que eu aprendesse tanto durante estes anos, à FCT e à Unidade de Investigação LASIGE, UID/CEC/00408/2013 que permitiram que este projeto se realizasse.

Não posso deixar de agradecer à minha família por todo o apoio ao longo destes anos. Aos meus avós, João e Carminda por todo o esforço que fizeram para que fosse possível concluir o curso. À minha mãe, Maria João por todo o apoio e incentivo que sempre me deu e também pela educação que me proporcionou que foi um fator determinante para que eu chegasse até aqui. Gostaria ainda de agradecer à minha irmã, Adriana, por nunca me ter deixado desanimar e por toda a cumplicidade.

Por fim, mas não menos importante gostaria de agradecer a todas as pessoas que me acompanharam neste percurso académico. Em especial, aos meus amigos Gustavo Correia, Pedro Caldeira e Miguel Viola que foram extremamente importantes ao longo deste percurso tanto nas horas de trabalho passadas em conjunto como nos grandes momentos de diversão. Muito obrigado!

Aos meus Avós.

Resumo

Os sistemas de software distribuídos têm uma comunicação intensiva onde a complexidade do padrão de mensagens trocadas entre processos tende a tornar a codificação dos mesmos difícil. As tecnologias existentes na área do software concorrente não são muito apropriadas ao desenvolvimento deste tipo de sistemas onde a comunicação é intensiva e bastante complexa devido ao elevado número de mensagens que são trocadas entre processos. É necessário definir mecanismos que permitam verificar se uma determinada sequência de interações feitas num canal de comunicação está bem estruturada e se está de acordo com um protocolo predefinido.

Os tipos de sessão foram desenvolvidos com o objetivo de colmatar as lacunas existentes no software concorrente com comunicação intensiva. Permitem definir protocolos que, recorrendo a tipos, representam uma "interação correta" do sistema e ainda garantem outras propriedades como a inexistência de erros na comunicação e de situações de impasse. Garantem também que uma mensagem é recebida ou que a comunicação num canal termina. Têm uma estrutura com recursividade terminal que implica que os protocolos que definem sejam descritos por uma linguagem regular.

No entanto, têm limitações na sua estrutura que impossibilitam a descrição eficiente de estruturas em forma de árvore. Os tipos de sessão livres do contexto foram apresentados como uma extensão dos tipos de sessão tradicionais e descrevem estruturas que não são possíveis de descrever recorrendo aos tipos de sessão tradicionais. Neste caso, os protocolos são descritos por linguagens determinísticas livres do contexto.

Deste trabalho resulta uma linguagem de programação chamada FreeST que é concorrente e explicitamente tipificada, onde os processos comunicam exclusivamente por troca de mensagens. Esta linguagem recorre a tipos de sessão independentes de contexto de modo a estender a expressividade dos tipos de sessão tradicionais e possibilitar a implementação, com segurança de tipos, de operações remotas em tipos de dados recursivos.

Palavras-chave: Linguagens de programação, concorrência, troca de mensagens, tipos de sessão, tipos de sessão independentes do contexto

Abstract

The society is highly dependent on software systems that are distributed and communication centered. The available tecnologies on this field are not well suited to develop systems with an intensive communication due to the large number of messages that are exchanged between processes. So, there is a need to establish protocols that are able to verify if a communication between parties is well formed in terms of the sequences of operations performed on a communication channel.

Session types were proposed as means to fulfil this kind of requirements such as defining the correct interaction or ensuring some safety properties like the abstence of communication errors and deadlocks. Liveness properties like the eventual receipt of a message or the termination of an interaction are also issues that can be addressed using these types. They describe structured interaction of processes on heterogeneously typed communication channels. This kind of types have a tail-recursive structure that imposes protocols that must be described by a regular language.

However, they have a limitation in their structure that makes it impossible to describe, in an efficient way, tree structured data. Context-free session types extend the notion of session types by allowing protocols that aren't tail-recursive and therefore are correspond to deterministic context-free languages.

With this work we aim to offer a concurrent typed programming language called FreeST where processes communicate exclusively by message passing. Using context-free session types we intend to have a programming language that is able to describe the low-level serialization of tree-structured data in a type safe way.

Keywords: Programming languages, concurrency, message passing, session types, context-free session types

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A engenharia de software moderna reconhece a importância das linguagens de programação com tipos bem definidos que procurem garantir que um sistema tem o comportamento esperado, isto é, sistemas cujo comportamento segue uma especificação. O principal propósito destes sistemas de tipos é prevenir que ocorram erros durante a execução de um programa. Assim sendo, as linguagens com tipos podem ajudar a definir sistemas em que se garante o comportamento esperado, na medida em que, fazem verificações (em tempo de compilação) com o intuito de averiguar se o sistema está de acordo com o sistema de tipos definido (*typechecking*).

A sociedade atual está bastante dependente de sistemas de software de larga escala, distribuídos e centrados na comunicação. As tecnologias existentes nas áreas de desenvolvimento de software concorrente não são muito apropriadas para o desenvolvimento de sistemas de comunicação intensiva, visto que têm falta de abstrações de alto nível para comunicações complexas.

No caso concreto do software concorrente, em que os processos comunicam por troca de mensagens, a comunicação torna-se facilmente complexa devido ao elevado número de mensagens que são trocadas entre os participantes.

Assim sendo, é importante definir abstrações que permitam controlar e estruturar a comunicação intensiva. Os tipos de sessão foram desenvolvidos com este propósito. Inicialmente foram propostos como uma extensão pi-calculus para especificar padrões de comunicação e verificar se a comunicação entre processos concorrentes está bem estruturada, foram depois alargados a outros tipos de linguagens incluindo as funcionais e as orientadas a objetos.

1.1 Motivação

A comunicação tem um papel central e de extrema importância. Os tipos de sessão são bastante úteis para sistemas que têm uma comunicação intensiva porque fornecem protocolos de comunicação através de tipos, segurança de tipos na comunicação (tipos

enviados e esperados coincidem) e, em alguns casos, garantias de que não ocorrem situações de impasse (*deadlocks*).

Os tipos de sessão têm inúmeras aplicações. Por exemplo, se for necessário enviar uma lista num canal (*stream*), podemos utilizar os tipos de sessão para garantir segurança de tipos na comunicação. O tipo de dados recursivo apresentado de seguida descreve uma lista:

```
type List = Nil | Cons int List
```

e o tipo de sessão, do ponto de vista de quem lê o canal, é o seguinte:

```
type ListServer = &{
  Nil : end
  Cons : ?int . ListServer
}
```

Neste tipo de sessão temos o operador & que oferece um ponto de escolha, ou seja, qualquer processo que leia de um canal do tipo ListServer tem de estar preparado para aceitar as duas opções especificadas: ou Nil e termina a comunicação ou Cons onde primeiro é lido um inteiro e de seguida é lida, recursivamente, a restante lista.

A sequência de operações para serializar a lista pode ser reconhecida por um autómato finito, apresentado abaixo:

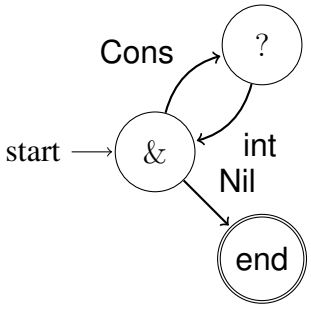


Figura 1.1: Autómato finito que representa a serialização de uma lista

Deste modo, podemos verificar que é possível enviar listas de um modo seguro recorrendo a tipos de sessão. Contudo, estes tipos têm limitações na sua estrutura que tornam impossível a descrição eficiente de serializações de dados estruturados em forma de árvore.

O exemplo proposto por Thiemann e Vasconcelos [20] mostra a limitação dos tipos de sessão, recorrendo ao envio de árvores binárias num canal de comunicação. Neste exemplo, à semelhança do anterior, definiu-se o tipo de dados para as árvores:

```
type Tree = Leaf | Node int Tree Tree
```

Para serializar esta estrutura é necessário percorrê-la numa determinada ordem transmitindo os valores das folhas (Leaf), dos nós (Node) e dos inteiros. A sequência de operações para serializar esta árvore pode ser descrita pela seguinte gramática livre do contexto:

```
N ::= Leaf | Node int N N
```

Se observarmos a linguagem produzida pelo não terminal N podemos constatar que é livre do contexto mas não é regular, contrariamente à linguagem do exemplo da lista, que é descrita pela seguinte expressão regular $(\&\mathsf{Cons}\ ?\mathsf{int})^*\ \&\mathsf{Nil}$. Como cada tipo de sessão tradicional tem como linguagem a união de uma linguagem regular com uma linguagem ω-regular, que descreve as sequências finitas e infinitas admitidas pelo tipo, podemos concluir que não é possível serializar uma árvore recorrendo a tipos de sessão.

Após apresentarem a limitação descrita no exemplo, Thiemann e Vasconcelos [20] introduziram os tipos de sessão livres do contexto. Nestes tipos é removida a continuação das primitivas de envio e de receção (por exemplo: S ::= ?T.S passa para S ::= ?T), é adicionado o operador de sequenciação (;) e a operação de skip que substitui a operação end. Deste modo, deixa de ser necessária a recursividade terminal que os tipos de sessão convencionais impõem e torna-se possível de descrever, comunicação livre do contexto como por exemplo, a sequência de operações que serializar uma árvore requer.

1.2 Contribuições

A principal contribuição deste trabalho é oferecer uma nova linguagem de programação funcional cuja sintaxe é semelhante à linguagem Haskell. As principais características da linguagem FreeST são:

- É concorrente, baseada em troca de mensagens em canais de comunicação síncronos e bidirecionais.
- É explicitamente tipificada, onde os processos comunicam exclusivamente por troca de mensagens.
- Os protocolos de troca de mensagens são definidos por tipos de sessão independentes de contexto.

1.3 Estrutura do documento

Este capítulo faz uma introdução ao nosso trabalho, apresentando as suas motivações e contribuições. Os capítulos seguintes estão organizados da seguinte forma:

- O capítulo 2 analisa brevemente as linguagens de programação baseadas em troca de mensagens (Go, FuSe e SePi), as incorporações de tipos de sessão regulares em Haskell e ainda as linguagens baseadas em atores (Erlang e Akka). Revê brevemente os conceitos de tipos de sessão e de modelo de atores.
- O capítulo 3 apresenta a linguagem FreeST. A linguagem é apresentada com recurso a exemplos que facilitam a apresentação da sintaxe e semântica e ainda do sistema de tipos e das expressões existentes.

- O capítulo 4 descreve as fases de validação e de interpretação de um programa e ainda a geração de código. Neste capítulo são ainda apresentados os testes realizados e os resultados obtidos.
- O capítulo 5 apresenta as conclusões obtidas e os planos para o trabalho futuro da linguagem.

Capítulo 2

Trabalho relacionado

Muitas linguagens de programação e outros formalismos foram sendo propostos ao longo do tempo para endereçar os aspetos de comunicação no software. Dois modelos comuns para lidar com a comunicação entre componentes numa computação concorrente são a memória partilhada e a troca de mensagens. No modelo em que a comunicação é feita através de memória partilhada os componentes interagem escrevendo e lendo em zonas de memória partilhadas enquanto que, no modelo de troca de mensagens os componentes comunicam enviando e recebendo mensagens através de um canal de comunicação.

O foco deste capítulo é descrever brevemente o modelo de troca de mensagens, destacando a programação baseada em canais de comunicação, descrita na secção 2.1 e a programação baseada em atores, descrita na secção 2.2. Este capítulo é composto pelas descrições sucintas dos conceitos de tipos de sessão (Secção 2.1.2) e do modelo de atores (Secção 2.2.1) e apresenta linguagens baseadas em canais de comunicação, Go (Secção 2.1.1), SePi (Secção 2.1.3) e FuSe (Secção 2.1.4) e linguagens baseadas em atores, Erlang (Secção 2.2.2) e Akka (Secção 2.2.3). Apresenta ainda a descrição da incorporação de tipos sessão regulares na linguagem Haskell.

O conteúdo presente neste capítulo pode ser completado pela referência [2] para os tipos de sessão, nomeadamente as aplicações nos diversos tipos de linguagens (orientadas a objetos, funcionais e imperativas). No que diz respeito a atores, o conteúdo pode ser complementado pelo livro [1].

2.1 Canais

Esta secção apresenta sucintamente as linguagens de programação Go e SePi, ambas baseadas em canais de comunicação. Apresenta também o conceito de tipos de sessão.

2.1.1 Go

A área do software concorrente tem sido alvo de um estudo intensivo ao longo do tempo. O mecanismo mais comum ao nível da comunicação é a memória partilhada que utiliza trincos

para garantir exclusão mútua em regiões críticas do programa. Alternativamente, Hoare apresentou a linguagem CSP (*Communicating Sequential Processes*) [13] que disponibiliza uma única primitiva: a comunicação síncrona. Nesta linguagem os processos comunicam através de canais cuja operação de envio bloqueia até que o recetor leia a mensagem, fornecendo assim um mecanismo de sincronização. Desta forma, Hoare alavancou o desenvolvimento de outras linguagens cujas primitivas de sincronização recorrem ao uso de canais.

Go ou *golang*, é uma linguagem de programação concorrente desenvolvida pela Google e que apareceu em 2009 [10]. É uma linguagem compilada, com verificação de tipos, baseada noutras linguagens como o Algol e o C.

A linguagem tem uma sintaxe semelhante à do C, mas com algumas alterações que procuram tornar o código mais conciso e legível, como é exemplo a combinação de declarações e inicializações: i := 23 (para uma atribuição usa-se o operador =) ou o facto das funções terem a possibilidade de retornar múltiplos valores.

Muitas vezes, os programadores têm dificuldade em programar em ambientes concorrentes devido às subtilezas que são inerentes ao uso de memória partilhada e ao acesso à mesma (uso de semáforos). É por este motivo que o Go promove uma programação de mais alto nível, recorrendo ao uso de canais, para enviar variáveis partilhadas. Assim sendo, não existe necessidade de recorrer a memória partilhada para partilhar os valores explicitamente pelos diferentes fios de execução.

As *goroutines* são funções que têm a capacidade de serem executadas concorrentemente com outras funções (rotinas go) no mesmo espaço de endereços. O custo associado à sua criação é bastante reduzido quando comparado com a criação de fios de execução tradicionais. Para criar este tido de rotinas é necessário preceder a invocação da função com a palavra reservada go.

O uso de canais permite que duas rotinas comuniquem entre elas e que sincronizem as suas execuções. O exemplo seguinte mostra como podem ser utilizados os canais de comunicação e as primitivas para envio e receção de mensagens:

```
1  func main() {
2    chInt := make(chan int)
3    chStr := make(chan string)
4    go server(chInt, chStr)
5
6    chInt <- 16
7    chInt <- 7
8
9    chStr <- "Hello "
10   chStr <- "World"
11
12   time.Sleep(10 * time.Millisecond)
13 }
```

```
14
15 func server (chInt <-chan int, chStr <-chan string){
16   for{
17     select {
18       case x := <-chInt :
19         fmt.Println(x + <-chInt)
20       case x := <-chStr :
21         fmt.Println(x + <-chStr)
22     }
23   }
24 }
```

Antes de um canal ser usado é necessário inicializá-lo e definir o tipo de dados que transporta, no exemplo acima podemos observar que o canal ch é inicializado como um canal onde são transportados inteiros: ch := **make** (**chan int**). As primitivas para enviar e receber valores de um canal são, respetivamente, ch <- 1 e a <- ch. As leituras bloqueiam até que seja lido um valor no canal e as escritas são, possivelmente bloqueantes até que outra rotina go leia os valores enviados.

O Go oferece uma abordagem para o desenvolvimento de software concorrente que disponibiliza simplicidade na escrita de sistemas estruturados com processos concorrentes.

2.1.2 Tipos de sessão

Os tipos de sessão apareceram, em primeiro lugar, numa variante do pi-calculus apresentando uma distinção sintática entre canais lineares e canais partilhados [14]. Mais tarde, Gay e Hole [8] introduziram o conceito de sub-tipos para os tipos de sessão o que permitiu que as especificações de um protocolo fossem estendidas levando a descrições mais ricas das interações.

Estes tipos têm como principal objetivo enriquecer a expressividade que os tipos tradicionais fornecem, tornando possível estruturar interações complexas numa computação concorrente onde são trocadas muitas mensagens em canais de tipos heterogéneos. As mensagens que são trocadas têm de ser recebidas na ordem certa (ordem em que foram enviadas) e têm de ser da natureza esperada. Por exemplo, suponhamos que temos dois processos que possuem duas extremidades distintas (X e Y) de um canal e que estão a comunicar por troca de mensagens. Se for escrito um inteiro na extremidade X, espera-se receber uma mensagem com esse inteiro e não com outro tipo de dados na extremidade Y. Da mesma forma, se for expectável que se escreva em X um inteiro e de seguida um booleano, em Y é expectável que as mensagens sejam recebidas por essa ordem. Se estas forem recebidas na ordem inversa (primeiro o booleano e depois o inteiro) temos uma quebra no protocolo pré-estabelecido. O tipo !integer.?boolean.end representa uma extremidade de um canal onde é escrito um inteiro, lido um booleano e de seguida termina a interação. O tipo ?integer.!boolean.end representa a extremidade contrária que tem

os tipos duais, ou seja, a que lê um inteiro, escreve um boleano e de seguida termina a interação.

Quando o objetivo do protocolo é oferecer em X um conjunto de opções à extremidade Y, então os seus tipos são, respetivamente, $\&\{l_i : T_i\}_{i\in I}$ e $\oplus\{l_i : T_i\}_{i\in I}$, nos quais o operador $\&$ representa a oferta de opções e o operador $\oplus$ representa a escolha de uma dessas opções. É ainda importante, garantir que não ocorrem situações de corrida, isto é, quando dois processos interagem os outros não podem interferir na comunicação.

A descrição deste tipo de protocolos, garantindo as condições supra-referidas, torna-se por vezes uma tarefa complexa e suscetível a erros por parte dos programadores e esse é um dos motivos que torna os tipos de sessão úteis.

Os tipos de sessão são usados para descrever interações entre exatamente dois fios de execução distintos, no entanto, há casos em que é requerido que uma das extremidades de um canal seja partilhada por vários fios de execução. Nesses casos, podemos ter canais lineares e canais partilhados.

Vasconcelos apresenta uma reconstrução dos tipos de sessão [21] na qual, baseado nas ideias de um sistema de tipos linear para o lambda calculus [22] propôs equipar os pré-tipos com a anotação *lin*, para obter tipos de sessão tradicionais e com a anotação *un*, para obter um canal que pode ser partilhado por um número indefinido de fios de execução.

2.1.3 SePi

Franco e Vasconcelos [7] apresentaram uma linguagem concorrente baseada no pi-calculus monádico onde as interações são feitas recorrendo a tipos que resultam de uma combinação entre tipos de sessão e tipos linearmente refinados.

Nesta linguagem, os canais de comunicação são síncronos e bi-direcionais. São descritos pelas suas duas extremidades, onde os processos podem ler ou escrever em qualquer parte dos programas e usam tipos para descrever o fluxo de mensagens que são escritas/lidas no canal.

Para declarar um canal onde apenas é trocado um inteiro e de seguida é terminada a interação entre os processos, basta invocar **new** w r : !**integer**.**end**, e ficamos na posse de duas variáveis. A variável *w* do tipo !**integer**.**end** e a variável *r* do tipo dual, ?**integer**.**end**.

```
1 type Session =
2     +{sumInt: !integer.!integer.end,
3       concatString: !string.!string.end}
4
5 new client server : *?Session
6
7 def serverProc () =
8   server!(new s : dualof Session).{
9     serverProc!() |
```

```
10      case s of
11          sumInt → s?x. s?y. printIntegerLn!x+y
12          concatString → s?x. s?y. printStringLn!x++y
13    }
14  serverProc!() |
15  client?c. c select sumInt. c!16. c!7 |
16  client?c. c select concatString. c!"Hello ". c!"World"
```

Se forem necessários canais cujas extremidades tenham de ser conhecidas por mais que um fio de execução é necessário torná-los partilhados, recorrendo à primitiva **un** (*unrestricted*), estes já não são livres de condições de corrida e é preciso continuar a garantir que a comunicação tem o fluxo de mensagens esperado e é livre de interferência.

Baltazar et al. introduziram um conceito em que combinam os tipos de sessão com alguns refinamentos originando assim os tipos linearmente refinados [4]. Estes tipos permitem ao programador acoplar formulas a tipos permitindo especificar propriedades nos tipos. Um exemplo de um tipo linearmente refinado é {x : **integer** | A}, no qual x representa um inteiro que tem de respeitar a formula A. Estes tipos serviram de base para implementar linguagens como o SePi [7].

Em SePi, o uso de recursos pode ser controlado com as primitivas de **assume** e **assert**. Estas primitivas são usadas, por exemplo, em transações bancárias, com o objetivo de garantir que não existem servidores a enganar clientes (cobrar um valor múltiplas vezes ou alterar o valor de uma cobrança antes de a comunicar ao banco). O cliente assume (**assume**) que o valor é para ser cobrado apenas uma vez e o banco, aquando da cobrança, verifica se foi dada permissão para cobrar aquele valor (**assert**). O problema desta solução é que o cliente e o banco são entidades completamente separadas (não se conhecem), logo não existe nenhuma maneira de garantir que os **assert** correspondem aos **assume** corretos. Usando tipos refinados podemos especificar o montante e o número de vezes que vai ser feita a cobrança. Um exemplo de tipos refinados para o caso dos bancos pode ser {x:**integer** | charge(ccard ,x)} [7]. Este tipo é interpretado como um inteiro para o qual a permissão charge(ccard ,x) foi emitida.

2.1.4 FuSe

A linguagem FuSe apresentada por Padovani [18] é uma implementação em OCaml com tipos de sessão que combina a execução estática de protocolos com verificações em tempo de execução sobre a linearidade das extremidades dos canais.

A inferência de tipos é indecidível quando estamos na presença de recursão polimórfica o que implica que seja necessário anotar as funções polimórficas recursivas com o seu tipo. O sistema de tipos da linguagem FuSe não contém o operador de sequenciação dos tipos de sessão independentes do contexto e introduz um combinador de ordem superior @> chamado *resumptions* para consumir um prefixo de um protocolo sequencial (T;S, por exemplo). A assinatura do combinador é @> : (T → 1) → T;S → S, ou seja, é uma função

que é aplicada à extremidade do canal de tipo T, realiza a comunicação e retorna o canal residual de tipo 1. No caso da extremidade do canal ser descrita por um tipo sequencial T;S, o combinador @> realiza a comunicação descrita pelo tipo prefixo T e termina deixando a extremidade do canal com o tipo da continuação S. Deste modo, é eliminada a necessidade de ter um sistema de tipos que dependa das anotações feitas pelo programador para as recursões polimórficas.

Contudo, esta aproximação tem limitações que são impostas pela necessidade dos processos operarem em extremidades dos canais que sejam duais. Por exemplo, um processo que opere na extremidade descrita pelo tipo (!**Int**;1);?**Bool** pode comunicar com outro processo que opere na extremidade com tipo (?**Int**;1) ;!**Bool** mas não com um processo cuja extremidade tenha o tipo !**Int**;?**Bool** apesar de (!**Int**;1);?**Bool** e !**Int**;?**Bool** serem equivalentes.

2.1.5 Tipos de sessão regulares em Haskell

Existem diversos trabalhos nos quais os tipos de sessão regulares são incorporados na linguagem Haskell. Neubauer e Thiemann [17] demonstram como incorporar tipos de sessão numa linguagem funcional cujo sistema de tipos seja suficientemente poderoso. Apresentaram uma implementação de uma biblioteca de tipos de sessão regulares em Haskell. Pucella e Tov [19] expõem uma alternativa diferente, na qual utilizam primitivas de concorrência que a linguagem Haskell disponibiliza.

Os trabalhos traduzem os tipos de sessão regulares para a linguagem Haskell utilizando técnicas avançadas da mesma. Contudo, quando são propostas bibliotecas que incorporam tipos de sessão numa linguagem levantam-se problemas de usabilidade. Na maioria dos casos tornam os tipos de sessão ilegíveis para os programadores, sobrecarregam a linguagem e tornam a codificação mais complexa. Outro problema que é comum às diferentes incorporações é a falta de precisão e de clareza das mensagens que reportam erros.

Relativamente aos tipos de sessão independentes do contexto não existe nenhum trabalho realizado através de incorporações, uma vez que as linguagens existentes não têm expressividade suficiente para o materializar.

Discussão Os tipos de sessão convencionais têm limitações na sua estrutura que impedem a serialização de forma eficiente e com segurança de tipos de estruturas de dados organizadas em forma de árvore, documentos XML, pilhas e de outros exemplos de estruturas não lineares. Estas limitações devem-se ao facto dos tipos de sessão convencionais apresentarem uma estrutura com recursividade terminal que apenas pode ser descrita por uma linguagem regular. As linguagens que descrevem as estruturas não lineares são independentes do contexto e como tal não é possível que os tipos de sessão convencionais descrevam a sua serialização. A linguagem que propomos permite definir protocolos que

descrevem eficientemente essas estruturas recorrendo a tipos de sessão independentes do contexto.

2.2 Atores

Esta secção apresenta brevemente o modelo de atores e a linguagem Erlang que implementa este modelo. Descreve ainda o Akka que é um conjunto de ferramentas que também implementa o modelo de atores e que pode ser utilizado nas linguagens de programação Java e Scala.

2.2.1 Modelo de atores

O conceito de atores foi introduzido por Hewitt et. al [11, 12] como um formalismo que se foca na relação entre eventos que é causada por um ator. O modelo de Hewitt [11] define uma série de construtores para um sistema de atores. Foi desenvolvida uma representação que esconde a informação interna para os sistemas de atores e que permite que estes se componham recorrendo à troca de mensagens [1]. Os problemas tradicionais que estão inerentes à programação concorrente como as situações de impasse e as condições de corrida foram também tidos em consideração [1], de modo a que não seja necessário recorrer ao uso de semáforos para limitar o acesso a regiões criticas dos programas.

Os atores são entidades concorrentes que trocam mensagens assincronamente. Cada ator tem um nome único e imutável, que pode ser comunicado, e um comportamento específico. Quando recebe uma mensagem, o seu comportamento pode ser um dos três seguintes: reencaminhar a mensagem para outros atores caso conheça os seus nomes, criar novos atores ou alterar ao seu comportamento que pode ou não ter efeito em comunicações futuras [1].

A semântica de um ator convencional fornece encapsulação, escalonamento justo, transparência na localização, localidade de referências e mobilidade [16].

O modelo de atores conta com as seguintes propriedades: cada ator apenas pode ter acesso ao seu estado interno (encapsulamento de estados), a informação só pode ser partilhada por troca de mensagens (não deve existir qualquer tipo de memória partilhada) e uma determinada mensagem é garantidamente entregue (escalonamento justo) a não ser que o ator que a ia receber tenha sido permanentemente desativado (consistência eventual).

2.2.2 Erlang

O Erlang é uma linguagem de programação que foi desenhada com o objetivo de desenvolver software concorrente, em tempo real e sistemas distribuídos tolerantes a faltas [3]. É a implementação mais conhecida do modelo de atores 2.2.1.

Os programadores de Erlang têm de especificar quais são as atividades que são representadas em processos paralelos e toda a comunicação existente entre os processos. Esta visão de concorrência é similar à do CSP [13] que visa obter o máximo desempenho compilando os programas para execução paralela e não para modelar a concorrência do mundo real.

O modelo de concorrência desta linguagem é baseado em processos com troca de mensagens assíncrona. Os mecanismos de concorrência são bastante simples e requerem pouco esforço computacional, uma vez que os processos precisam de pouca memória tanto para serem criados, destruídos e ainda para comunicar via troca de mensagens.

Para criar e começar a execução de um novo processo é necessário invocar a função **spawn** que em vez de avaliar e retornar o resultado da função retorna o **Pid** (identificador do processo) A única maneira de comunicar entre processos é por troca de mensagens. Para enviar uma mensagem de um processo para outro usa-se a primitiva '!', por exemplo, `A ! Message1` (envia uma mensagem para o processo com o identificador Pid1). O outro processo para receber a mensagem usa a primitiva **receive**.

O exemplo seguinte mostra como enviar e receber mensagens em Erlang assim como a utilização da primitiva **spawn** para criar e executar processos:

```
1  server () →
2    receive
3      {addInt,X,Y} →
4        io:format("~p~n", [X+Y]),
5        server();
6      {concatStr, X,Y} →
7        io:format("~p~n", [string:concat(X,Y)]),
8        server()
9    end.
10
11 start () →
12   PID = spawn(?MODULE, server, []),
13   clients(PID).
14
15 clients(ServerPID) →
16   ServerPID ! {addInt, 16,7},
17   ServerPID ! {concatStr,"Hello ","World"}.
```

No exemplo acima, {addInt,X,Y} e {concatStr, X,Y} são padrões usados para associar uma mensagem recebida a um determinado comportamento.

Neste modelo, as mensagens são sempre entregues aos processos. Cada processo tem uma "caixa de correio" (*mailbox*) onde armazena as mensagens que vão sendo recebidas por ordem de chegada (igual à ordem de envio), à semelhança do modelo de atores brevemente descrito na secção 2.2.1.

2.2.3 Akka

Akka é um conjunto de ferramentas que foi desenhado com o intuito de construir sistemas escaláveis e resilientes. O objetivo principal é fornecer ferramentas que possibilitem que o programador se foque nos requisitos do sistema e não nas particularidades de baixo nível como garantir comportamento confiável, tolerância a faltas e desempenho.

O Akka implementa o modelo de atores proposto por Hewitt [12] com o objetivo de fornecer um nível de abstração que facilite a escrita de sistemas concorrentes, paralelos e distribuídos, usufrui do facto de poder ser utilizado em Java (também em Scala) que é uma das linguagens de programação mais utilizadas. Uma das implementações mais conhecidas deste modelo foi o Erlang 2.2.2 que apesar de usufruir de todas as vantagens para o desenvolvimento de software concorrente que o modelo oferece, não teve uma adoção tão geral como o Java.

Em Akka, os atores são semelhantes aos descritos em 2.2.1: são assíncronos e usam um sistema de comunicação baseado em troca de mensagens não bloqueantes, são considerados processos mais leves, são orientados a eventos, isto é, esperam por mensagens e de seguida reagem a essas mensagens. As mensagens que transitam entre atores não têm tipos apesar de existirem atores com tipos nas versões mais recentes.

Um ator tem estado, uma "caixa de correio" (*mailbox*) e um determinado comportamento, pode ter como filhos outros atores por ele criados e pode definir uma estratégia para lidar com as suas faltas (estratégia de supervisor). Ao contrário dos objetos do Java que são destruídos pelo coletor de lixo quando deixam de ser referenciados, os atores têm de ser explicitamente terminados (context.**stop**(actorRef)). O comportamento dos atores pode mudar ao longo do tempo (operações **become** e **unbecome**).

As mensagens que estão à espera de ser processadas são adicionadas à "caixa de correio" e são tratadas de acordo com um determinado critério. A política usada por defeito para tratamento das mensagens é a política FIFO, contudo existe a possibilidade de definir políticas alternativas. O comportamento que está em vigor tem de tratar sempre da mensagem seguinte, visto que, não existe a possibilidade de procurar mensagens na "caixa de correio" como existe em Erlang.

O exemplo seguinte apresenta parte do código que ilustra uma interação simples entre atores em Akka (escrito em Java) na qual é enviada uma mensagem para o ator que soma dois inteiros ou concatena duas strings. O código completo encontra-se no apêndice A.

```
1 public class Main {
2   public static void main(String[] args) {
3     ActorSystem sys = ActorSystem.create("system");
4
5     ActorRef server = sys.actorOf(Props.create(Server.class));
6
7     server.tell(new IntMessage(16, 7), ActorRef.noSender());
```

```
8       server.tell(new StringMessage("Hello ", "World"), ActorRef.
            noSender());
9     }
10  }
11
12  public class Server extends AbstractActor{
13    @Override
14    public Receive createReceive() {
15      return new ReceiveBuilder()
16          .match(IntMessage.class, msg → {
17            System.out.println(msg.getFst() + msg.getSnd());

18          })
19          .match(StringMessage.class, msg → {
20            System.out.println(msg.getFst() + msg.getSnd());

21          })
22          .build();
23    }
24  }
```

Discussão O modelo de atores foca-se no conceito de ator e na relação entre os eventos causada pelos atores. Tem em consideração aspetos relativos à programação concorrente como as situações de impasse e as condições de corrida que também são tidos em consideração na programação baseada em canais. A diferença em relação à linguagem que apresentamos é que os atores trocam mensagens de forma assíncrona enquanto que a nesta linguagem as mensagens são trocadas de forma síncrona.

Capítulo 3

A Linguagem FreeST

Este capítulo apresenta a linguagem FreeST, a sua sintaxe, sistema de tipos e semântica. A secção 3.1 apresenta a sintaxe da linguagem descrevendo os seus tipos, os *kinds*, as expressões, as funções e as declarações de tipos de dados. A secção 3.2 apresenta os sistemas que permitem garantir que um programa está bem formado.

3.1 A linguagem

FreeST é uma linguagem funcional que tem uma sintaxe semelhante à da linguagem Haskell, acrescida de primitivas que permitem criar canais e enviar e receber valores nos mesmos. As primitivas de comunicação disponibilizadas pela linguagem são a troca de mensagens e as escolhas. As mensagens são trocadas em canais de comunicação síncronos e bidirecionais. Cada canal pode ser descrito pelas suas duas extremidades (*endpoints*) e caracterizado por tipos que descrevem o padrão de mensagens que passam no canal. Os processos podem ler ou escrever em cada uma das extremidades que compõem o canal.

3.1.1 Tipos

B	$::= \mathbf{Int} \mid \mathbf{Char} \mid \mathbf{Bool} \mid ()$	Tipos básicos
T	$::= \mathbf{Skip} \mid T;T \mid \,!B \mid ?B$	Tipos
	$\mid \oplus\{l_i{:}T_i\}_{i\in I} \mid \&\{l_i{:}T_i\}_{i\in I}$	
	$\mid B \mid T \to T \mid T \multimap T$	
	$\mid (T,T) \mid [l_i{:}T_i]_{i\in I} \mid \mathbf{rec}\,\alpha\,.\,T \mid \alpha$	
$\mathcal{C}$	$::= T \mid \mathbf{forall}\,\alpha \Rightarrow \mathcal{C}$	Esquemas de tipos

Figura 3.1: Sintaxe dos tipos

A figura 3.1 descreve a linguagem dos tipos disponibilizados pela linguagem. Os tipos incluem tipos primitivos, denotados por B. Estes são parte dos tipos primitivos da linguagem Haskell e incluem números inteiros (**Int**), caracteres (**Char**), booleanos (**Bool**) e o tipo unitário $()$. Os restantes tipos disponíveis são compostos pelo operador de sequenciação $T_1;T_2$ que descreve a extremidade de um canal que se comporta primeiro segundo T_1 e depois segundo T_2, pelo tipo **Skip** que é a unidade do operador de sequenciação e que indica que não há comunicação, pelos tipos que representam o envio e receção de valores num canal: $!B$ e $?B$ respetivamente, pelas escolhas que podem ser tanto internas $\oplus\{l_i{:}T_i\}_{i\in I}$ que selecionam e enviam uma etiqueta num canal, como externas $\&\{l_i{:}T_i\}_{i\in I}$ que disponibilizam uma escolha de uma etiqueta (um dos ramos) e depois continuam nesse ramo. Estão ainda disponíveis funções lineares $T \multimap T$ e funções convencionais (partilhados ou *unrestricted*) $T \to T$ assim como os tipos de dados tradicionais $[l_i{:}T_i]_{i\in I}$ das linguagens funcionais, os tipos recursivos $\mathbf{rec}\,\alpha\,.\,T$ e ainda as variáveis de tipo α.

Assim sendo, um tipo **!Int**;**?Bool** é um tipo que descreve uma extremidade de um canal cujo protocolo consiste em enviar um número inteiro, receber um valor booleano e de seguida terminar sem mais nenhuma interação.

Para além dos tipos acima descritos, a figura 3.1 apresenta ainda os esquemas de tipos cuja sintaxe é **forall** x ⇒ T. A linguagem proposta por Thiemann e Vasconcelos [20] inclui os esquemas de tipos $\mathcal{C}$ nos tipos T enquanto que a linguagem FreeST separa as duas definições com o objetivo de simplificar a fase de validação do compilador, visto que, os esquemas de tipos apenas podem aparecer no *top-level*, isto é, nas declarações de tipos de cada função (ex: id :: **forall** a ⇒ a → a).

3.1.2 Géneros

υ	$::= \mathcal{S} \mid \mathcal{T}$	$\mathcal{S} < \mathcal{T}$	*Prekinds*
m	$::= \mathbf{u} \mid \mathbf{l}$	$\mathbf{u} < \mathbf{l}$	Multiplicidades
κ	$::= \upsilon^m$		*Kinds*
Δ	$::= \varepsilon \mid \Delta, \alpha :: \kappa$		Ambientes de *kinding*

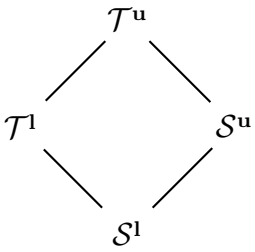
Figura 3.2: Sintaxe dos *kinds* e dos ambientes de *kinding*

A figura 3.2 apresenta a sintaxe dos géneros (*kinds*) que são compostos por um *prekind* υ e por uma multiplicidade m e são representados por υ^m. Os *prekinds* υ podem ser um de dois tipos distintos: tipos de sessão $\mathcal{S}$ ou tipos funcionais $\mathcal{T}$; e as multiplicidades m podem ser lineares $\mathbf{l}$ ou partilhado $\mathbf{u}$ (*unrestricted*).

A relação de ordem entre *prekinds* é dada pela regra $\mathcal{S} < \mathcal{T}$. Esta relação descreve que um tipo de sessão $\mathcal{S}$ pode ser visto como um tipo funcional $\mathcal{T}$. No que diz respeito

às multiplicidades, a relação de ordem é dada pela regra $\mathbf{u} < \mathbf{l}$, exprimindo que uma multiplicidade linear pode ser julgada como uma multiplicidade de um tipo partilhado. Consequentemente, a relação entre géneros é dada pela regra $\upsilon_1^{m_1} \leq \upsilon_2^{m_2}$ se e só se $\upsilon_1 \leq \upsilon_2$ e $m_1 \leq m_2$.

Os ambientes de *kinding* Δ (representados na figura 3.2) são compostos por associações de variáveis α a géneros κ.



Como referido anteriormente, na linguagem proposta por Thiemann e Vasconcelos [20] os esquemas de tipos $\mathcal{C}$ estão incluídos na definição de tipos T, portanto é necessário que os *prekinds* tenham na sua definição uma categoria para estes tipos $\mathcal{C}$. Como na linguagem FreeST os esquemas de tipos são uma categoria diferente não há necessidade que a definição de género inclua uma categoria para os classificar.

3.1.3 Expressões

$e ::= () \mid n \mid c \mid \mathbf{True} \mid \mathbf{False}$	Expressões básicas
$\mid x \mid \mathbf{let}\ x = e\ \mathbf{in}\ e$	Variáveis e Let
$\mid ee \mid e\,[T]$	Aplicações
$\mid \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e$	Condicional
$\mid (e, e) \mid \mathbf{let}\ x, y = e\ \mathbf{in}\ e$	Pares
$\mid \mathbf{new}\ T \mid \mathbf{send}\ e\ e \mid \mathbf{receive}\ e$	Operações sobre canais
$\mid \mathbf{select}\ C\ e \mid \mathbf{match}\ e\ \mathbf{with}\ \{C_i \rightarrow e_i\}_{i \in I}$	
$\mid \mathbf{fork}\ e$	Fork
$\mid C \mid \mathbf{case}\ e\ \mathbf{of}\ \{C_i \rightarrow e_i\}_{i \in I}$	Tipos de dados

Figura 3.3: Sintaxe das expressões

A figura 3.3 apresenta a sintaxe das expressões que estão disponíveis na linguagem FreeST. As expressões básicas são os inteiros n (ex: 1), caracteres c (ex: 'a'), os booleanos (**True** e **False**) e ainda o valor do tipo unitário ().

As expressões também podem ser variáveis x ou associações de valores de expressões a variáveis com o operador **let** x = e **in** f. Os valores associados x = e apenas são acessíveis na expressão f, a seguir à palavra reservada **in**. Por exemplo, a expressão **let** x = 5 **in** 10*x associa o valor 5 à variável x no âmbito da segunda expressão, ou seja, a expressão 10*x.

Nesta linguagem estão disponíveis aplicações de expressões e aplicações de tipos. As aplicações de expressões podem ser, por exemplo, chamadas a funções como no seguinte caso f x. As aplicações de tipos estão disponíveis com a sintaxe e[T], onde T é aplicado

à expressão e. Neste caso, é expectável que e tenha um tipo polimórfico (**forall** x ⇒ U) para que seja possível substituir as ocorrências da variável do esquema de tipos x no tipo U pelo tipo T.

A expressão condicional é igual à da linguagem Haskell cuja sintaxe é **if** b **then** e1 **else** e2: a expressão b é avaliada para um valor do tipo booleano que, no caso de ser **True** avalia a expressão e1 e no caso de ser **False** avalia a expressão e2.

Na linguagem FreeST existem duas expressões que envolvem pares de valores. O construtor de pares (e, f) e o destrutor de pares **let** x, y = e **in** f que é semelhante à operação de **let** previamente apresentada, mas é esperado que a primeira expressão seja um par para que os seus elementos sejam respetivamente associados às variáveis x e y que estão disponíveis apenas na segunda expressão, f.

Dado que os processos comunicam exclusivamente por troca de mensagens em canais são necessárias operações para operar sobre os canais. As operações que estão disponíveis são o **new** T, o **send** e f, o **receive** e, o **select** e e o **match** e **with {**...**}**. A expressão **new** T é responsável pela criação de novos canais e devolve um par que contém as duas extremidades que descrevem o canal. O primeiro elemento do par é a extremidade que é descrita pelo tipo T (primeiro elemento do par) enquanto que, o segundo elemento representa a outra extremidade que é descrita pelo tipo dual de T, isto é, o tipo com as operações de T invertidas. Por exemplo, se considerarmos o tipo ?**Int**;**!Bool**, o seu tipo dual é **!Int**;?**Bool**. Se o tipo for +**{**A:**!Int**, B:**Skip}** o tipo dual é &**{**A:?**Int**, B:**Skip}**.

As expressões de envio e de receção de valores (**send** e **receive**) concretizam a troca de mensagens. O processo responsável pelo envio escreve um valor v numa das extremidades do canal, enquanto que o processo que recebe lê na outra extremidade. O resultado de enviar um valor na extremidade a de um canal é o próprio b. Por outro lado, o resultado de receber um valor na extremidade b do canal é o par (v, b), em que v é o valor recebido. Como ambas as operações devolvem as extremidades a e b, ambos os processos podem continuar a interação sobre o canal.

As operações de **select** e **match** são similares às operações de envio e de receção de valores. A operação **select** l permite selecionar uma opção através de uma etiqueta l numa das extremidades do canal, enquanto que, a operação **match** e **with {**li→ ei**}** disponibiliza as opções que podem ser selecionadas na outra extremidade do canal.

A expressão **fork** e permite criar novos fios de execução (*threads*) que computam a expressão passada como parâmetro e, concorrentemente com os demais fios de execução. A expressão **fork** e avalia para (), o valor do tipo unitário.

As expressões que manipulam os tipos de dados são os construtores C e o **case** e **of {** $C_i \rightarrow e_i$**}** que é o operador tradicional que aceita uma expressão e executa blocos de código específicos com base nos valores dessa expressão.

3.1.4 Funções, declarações de tipos de dados e programas

Declarações de tipos de dados

A linguagem permite definir tipos de dados semelhantes aos da linguagem Haskell que assumem a seguinte forma:

$$\textbf{data}\ \mathrm{D} = (C_1\ \ T_{1_0}\ \ ...\ \ T_{n_0} \mid ... \mid C_k\ \ T_{1_k}\ ...\ T_{n_k})^{m}_{k \geq 1, n \geq 0}$$

A declaração de novos tipos de dados é semelhante à que pode ser encontrada na linguagem Haskell e é introduzida pela palavra reservada **data** que tem de ser sucedida de um construtor D (um identificador que começa por uma letra maiúscula).

Um exemplo de definição de um tipo de dados, é a lista de inteiros:

```
data IntList = Nil | Cons Int IntList
```

A definição de uma IntList é representada por dois construtores, Nil e Cons. O primeiro, não tem nenhum tipo como parâmetro enquanto que, o segundo tem um inteiro (**Int**) e uma IntList como parâmetros.

Funções

Para definir uma função é necessário definir o seu tipo (a assinatura) e declarar a função (corpo da função). O tipo de uma função e a sua declaração não têm uma ordem específica, isto é, a declaração do tipo pode ser feita antes ou depois da declaração da função (desde que não existam funções ou declarações de tipos duplicadas ou tipos sem funções ou ainda funções sem tipos).

Para declarar um tipo de uma função é necessário definir o seu nome (único para evitar duplicados), seguido de :: e de um esquema de tipos ($\mathcal{C}$) . Assim sendo, um exemplo de um tipo de uma função identidade seria id :: **forall** a ⇒ a → a ou ainda o caso de uma função não polimórfica que representa a soma dois inteiros: sum :: **Int** → **Int** → **Int**.

A declaração de uma função tem a forma fun x1 ... xn = e. Mais precisamente, em primeiro lugar é necessário definir o seu nome, de seguida um número variável de parâmetros e no fim, após o sinal de igualdade, a expressão que determina o seu comportamento. A implementação tradicional da função id para a qual definimos o tipo é: id x = x.[1]

Programas

Um programa p é constituído por declarações de tipos de dados, declarações de tipos de funções e declarações de funções como indica a figura 3.4.

[1] A função definida nos exemplos anteriores tem o nome id. Se fosse realmente necessário implementar esta função, o nome teria de ser alterado por exemplo para id'. Esta necessidade deve-se ao facto de o *prelude* da linguagem Haskell estar disponível e de conter uma definição da função id.

$$
\begin{aligned}
p \ ::= & \text{ Declaração de tipos de dados} \\
& \mid \text{Declaração de tipos de funções} \\
& \mid \text{Declaração de funções}
\end{aligned}
$$

Figura 3.4: Sintaxe de um programa

Um programa bem formado obedece a certas restrições tanto ao nível das funções definidas como ao nível das declarações de tipos de dados.

No que diz respeito às funções, como já foi referido anteriormente, todas as declarações de tipos de funções têm de ter uma declaração de uma função associada assim como a situação inversa, isto é, todas as funções têm de ter o seu tipo declarado. Para além destas restrições, também é obrigatório que a função `start` esteja definida (função principal do programa) porque esta é utilizada no processo de tradução de código e ainda que a função `main` não esteja definida para que não não existam conflitos aquando da tradução para a linguagem Haskell.

As restrições das declarações de tipos de dados são as mesmas que são impostas noutras linguagens como por exemplo as da linguagem Haskell. Não podem ser declarados tipos de dados com o mesmo nome, assim como, não podem existir construtores com o mesmo nome, mesmo que sejam definidos em declarações de tipos diferentes.

3.1.5 Explicação da linguagem com recurso a um exemplo

Ao longo desta secção vamos considerar como exemplo a transmissão de um tipo de dados estruturado na forma uma árvore binária de inteiros num único canal de comunicação. Este exemplo tem como objetivo apresentar uma explicação mais detalhada da linguagem.

Podemos dividir este exemplo em quatro partes distintas: definição do tipo de dados que representa uma árvore binária, envio da árvore binária, receção da árvore binária e definição da função `start` que declara uma árvore, cria os canais para a comunicação, envia e recebe a árvore em fios de execução separados e retorna a árvore recebida.

Tipo de dados de uma árvore O tipo de dados que descreve uma árvore binária de inteiros pode ser declarado da seguinte forma:

```
data Tree = Leaf | Node Int Tree Tree
```

Este tipo de dados define que os constituintes de uma árvore são folhas (`Leaf`) ou nós da árvore (`Node`) que têm na sua estrutura um valor inteiro (`Int`) e duas subárvores (`Tree`).

Envio da árvore binária Para enviar uma árvore de inteiros é necessário definir o tipo da função que representa o envio da árvore e de seguida define-se a função que concretiza o envio. O tipo de sessão que descreve o envio de uma árvore binária de inteiros é **rec** x . +{LeafC: **Skip**, NodeC: **!Int** ;x;x}. Este tipo, apresenta uma escolha interna ($\oplus\{l_i{:}T_i\}_{i\in I}$) que é composta por duas opções distintas: Leaf e Node. A primeira opção, LeafC : **Skip**, tem o tipo **Skip** que representa um canal residual, ou seja, onde não há comunicação. A segunda opção (NodeC) tem o tipo **!Int** ;x;x que é composto por um tipo sequencial, no qual podemos verificar que primeiro é enviado um inteiro (valor de cada nó da árvore) e de seguida são feitas duas chamadas recursivas do tipo representadas para variável x que permitem que se enviem as duas subárvores que cada Node contém. A assinatura da função que envia uma árvore é:

```
sendTree :: forall a ⇒ Tree → (rec x . +{LeafC : Skip, NodeC: !
    Int;x;x}); a → a
```

A função recebe uma Tree e um canal que primeiro corre o protocolo recursivo **rec** x . +{LeafC : **Skip**, NodeC: **!Int**;x;x} e de seguida o protocolo especificado por a. O canal especificado por a é depois retornado o que permite deixar o seu processamento para a continuação. Esta abstração é absolutamente necessária porque com um tipo sem a abstração: Tree → **rec** x . +{LeafC : **Skip**, NodeC: **!Int**;x;x} → **Skip** não seria possível continuar a escrita da árvore após a primeira chamada recursiva (para a sub-árvore esquerda). Isto verifica-se porque o tipo em questão termina em **Skip** e não tem a continuação descrita por a após o protocolo recursivo, implicando assim que após a primeira chamada recursiva seja retornado **Skip** e como este indica que não há comunicação a escrita terminaria, sem nunca enviar a sub-árvore direita.

A partir deste momento iremos abreviar o tipo recursivo da função (sendTree) **rec** x . +{LeafC : **Skip**, NodeC: **!Int**;x;x} para TreeChannel de modo a facilitar a leitura do exemplo.

O segundo passo é definir a função que concretiza o envio da árvore. No caso em que se envia uma Leaf, é necessário escolher o ramo através da operação **select** LeafC que espera um canal que tenha tipo na forma $\oplus\{l_i{:}T_i\}_{i\in I}$ (canal especificado no tipo da função). Por outro lado, quando se envia um Node, também é necessário selecionar o ramo certo através da operação **select** NodeC. Após esta operação, é necessário cumprir o protocolo que o tipo **!Int** ;TreeChannel;TreeChannel descreve. Assim sendo, primeiro é necessário enviar o inteiro v no canal c (**send** v c) e de seguida, é necessário fazer uma chamada recursiva à função para as duas subárvores do nó.

O código da função que envia árvores binárias é apresentada na listagem 3.1. Os parâmetros t e c são respetivamente a árvore a enviar e o canal onde enviar a árvore. A expressão **case** separa o código necessário para enviar uma Leaf ou um Node com base no valor da árvore t.

```
1  sendTree :: forall a ⇒ Tree → TreeChannel; a → a
2  sendTree t c =
3   case t of
4     Leaf → select LeafC c
5     Node x l r →
6       let c1 = select NodeC c in
7                                -- c1: !Int;TreeChannel;TreeChannel;a
8       let c2 = send x c1 in
9                                -- c2: TreeChannel;TreeChannel;a
10      let c3 = sendTree[TreeChannel;a] l c2 in
11                               -- c3: TreeChannel;a
12      let c4 = sendTree[a] r c3 in
13                               -- c4: a
14      c4
```

Listing 3.1: Envio de uma árvore binária

As operações **select** e **send** devolvem o mesmo canal utilizado na operação mas com um tipo diferente, isto é, **select** NodeC (linha 6) devolve um canal com o tipo **!Int**;TreeChannel;TreeChannel;a, originando um novo identificador c1 que é introduzido pelo operador **let**. A operação de **send** devolve um canal com um tipo igual mas sem a operação de **!**B. Neste caso, a expressão **send** x c1 (linha 8) envia o valor x no canal c1 cujos tipos são **Int** e **!Int**;TreeChannel;TreeChannel;a respetivamente. O tipo resultante desta operação é TreeChannel;TreeChannel;a, visto que o tipo **!Int** é consumido por esta operação.

As chamadas a funções polimórficas são da forma e**[**T**]** porque nesta linguagem é necessário especificar o tipo que as variáveis vão adotar em cada chamada à função. Neste exemplo, as chamadas da função sendTree especificam os tipos que variável a vai adotar. Na primeira chamada (linha 10) é feita uma aplicação de tipos à função sendTree com o tipo TreeChannel;a para que o tipo do parâmetro c fique (TreeChannel;a) [TreeChannel ;a/a] que é igual a TreeChannel;TreeChannel;a. Na segunda, a aplicação é feita com o tipo a para que c fique com o tipo (TreeChannel;a)[a/a] que é igual ao tipo TreeChannel;a que o tipo necessário para envia a sub-árvore direita. Assim sendo, a função sendTree termina com o tipo a como descrito na sua assinatura. Sempre que uma função com tipo é polimórfico é chamada é necessário que seja através de uma aplicação de tipos para que no tipo da função as variáveis polimórficas sejam substituídas por tipos concretos, eliminando assim a necessidade de fazer inferência de tipos (em casos como este).

Receção da árvore binária A função que recebe a árvore binária, apresentada na listagem 3.2, é análoga mas com o tipo de sessão dual **rec** x . **&{**Leaf: **Skip**, Node: **?Int**;x;x**}** (TreeChannelR daqui em diante) que, em vez de impor a seleção de um dos ramos através da expressão **select** l, oferece uma escolha **match** tree **with {**$l_i \rightarrow C_i$**}** dos mesmos.

```
11 receiveTree :: forall a ⇒ TreeChannelR; a → (Tree, a)
12 receiveTree c =
13   match c with
14     LeafC c1 → (Leaf, c1)
15     NodeC c1 →
16                           -- c1: ?Int;TreeChannelR;TreeChannelR;a
17       let x, c2 = receive c1 in
18                           -- c2: TreeChannelR;TreeChannelR;a
19       let left, c3 = receiveTree [TreeChannelR;a] c2 in
20                           -- c3: TreeChannelR;a
21       let right, c4 = receiveTree [a] c3 in
22                           -- c4: a
23       (Node x left right, c4)
```

Listing 3.2: Receção de uma árvore binária

Esta função utiliza a expressão match para adotar um determinado comportamento consoante a etiqueta (linhas 14 e 15–19) que é escolhida pela operação **select**. A função devolve um par de valores com a árvore recebida e com o canal que permite continuar a leitura.

A operação de **receive** (linha 16) devolve um par de valores no qual o primeiro elemento é o valor recebido e o segundo é o mesmo canal que, à semelhança da operação de **send**, tem um tipo diferente. Neste exemplo, o tipo do canal c1 (linha 16) é **?Int** ;TreeChannelR;TreeChannelR;a portanto, o tipo do canal (c2) resultante da operação **receive** c1 será TreeChannelR;TreeChannelR;a, porque esta operação consome o tipo que descreve a receção de dados (**?Int**).

Função start A função apresentada no listagem 3.3 é a função principal do programa que é responsável por criar uma árvore, denominada por inTree (linha 22), criar o canal de comunicação (linha 23) que é descrito pelas suas duas extremidades writer e reader respetivamente. De seguida envia e recebe a árvore, concorrentemente, em dois fios de execução separados (linhas 24 e 25). Finalmente devolve a árvore que foi recebida (linha 26).

A expressão **new** TreeChannel cria um canal e devolve um par de valores com as duas extremidades do canal que são associadas às variáveis writer e reader pelo destrutor de pares (**let** writer, reader). O tipo da primeira extremidade é o tipo definido depois do operador **new** enquanto que o tipo da segunda extremidade é o seu dual TreeChannelR que são respetivamente os tipos dos canais que as funções sendTree (listagem 3.1) e receiveTree (listagem 3.2) esperam receber. A expressão **fork** devolve o valor unitário () e chama a função sendTree concorrentemente. A função receiveTree recebe a árvore que fica associada à variável outTree e que é depois devolvida.

O tipo **Skip** é aplicado às funções sendTree (linha 24) e receiveTree (linha 25) porque

este para além de representar um canal onde não há comunicação, representa um canal residual, isto é, um canal que foi consumido até ao fim. Com esta aplicação garantimos que quando a comunicação termina a árvore foi toda enviada.

```
20 start :: Tree
21 start =
22   let inTree = Node 7 (Node 5 Leaf Leaf) (Node 9 (Node 11 Leaf
       Leaf) (Node 15 Leaf Leaf)) in
23   let writer, reader = new TreeChannel in
24   let v = fork (sendTree[Skip] inTree writer) in
25   let outTree, r = receiveTree[Skip] reader in
26   outTree
```

Listing 3.3: Função principal para a transmissão de uma árvore binária

3.2 Validação

Para garantir que o sistema está isento de erros a fase de validação é composta por um sistema de *kinding* que procura verificar se todos os tipos estão bem formados. Na secção 3.2.1 são apresentadas as regras que compõem este sistema. É também feita uma verificação de tipos que está descrita na secção 3.2.3 e procura assegurar que todas as expressões têm o tipo esperado. Para efetuar estas verificações é, por vezes, necessário determinar quando é que dois tipos são equivalentes. A relação de equivalência é descrita na secção 3.2.2.

3.2.1 Sistema de *kinding* algorítmico

O sistema de *kinding* tem como objetivo determinar se um tipo está bem formado. Este sistema, distingue tipos de sessão de tipos funcionais e ainda diferencia tipos lineares de tipos convencionais (*unrestricted*) cuja diferença é que os primeiros podem ser utilizados zero ou mais vezes, enquanto que, os segundos (lineares) têm de ser utilizados exatamente uma vez. É definido pela avaliação da função $\Delta \vdash_a T \to \kappa$ cujas regras estão presentes na figura 3.5.

$$\frac{\overbrace{\ldots}^{\text{Específico para cada expressão e}}}{\underbrace{\Delta \vdash_a T}_{\text{In}} \to \underbrace{\kappa}_{\text{Out}}}$$

Segundo as regras apresentadas na figura 3.5, um tipo de sessão pode ser **Skip** cuja multiplicidade é $\mathbf{u}$, ou ainda os tipos que representam o envio e a receção de valores num canal ($!B$ ou $?B$) ambos com multiplicidades $\mathbf{l}$. Os restantes tipos de sessão são as escolhas, tanto internas $\oplus$ como externas $\&$ também com multiplicidade $\mathbf{l}$, e ainda o

$$\frac{}{\Delta \vdash_a \mathbf{Skip} \to \mathcal{S}^{\mathbf{u}}} \quad \frac{}{\Delta \vdash_a\ !B \to \mathcal{S}^{\mathbf{l}}} \quad \frac{}{\Delta \vdash_a ?B \to \mathcal{S}^{\mathbf{l}}} \quad \frac{}{\Delta \vdash_a\ B \to \mathcal{T}^{\mathbf{u}}}$$

$$\frac{x :: \kappa \in \Delta}{\Delta \vdash_a x \to \kappa} \quad \frac{\Delta \vdash_a T_1 \to \kappa_1 \quad \Delta \vdash_a T_2 \to \kappa_2}{\Delta \vdash_a T_1 \to T_2 \to \mathcal{T}^{\mathbf{u}}} \quad \frac{\Delta \vdash_a T_1 \to \kappa_1 \quad \Delta \vdash_a T_2 \to \kappa_2}{\Delta \vdash_a T_1 \multimap T_2 \to \mathcal{T}^{\mathbf{l}}}$$

$$\frac{\Delta \vdash_a T_1 \to \kappa_1 \quad \Delta \vdash_a T_2 \to \kappa_2}{\Delta \vdash_a (T_1, T_2) \to \mathcal{T}^{\mathbf{l}}} \quad \frac{\Delta \vdash_a T_1 \to \kappa_1 \ldots \Delta \vdash_a T_n \to \kappa_n}{\Delta \vdash_a [l_i{:}T_i]_{i \in I} \to \mathcal{T}^{\max(\kappa_1 \ldots \kappa_n)}}$$

$$\frac{\Delta \vdash_a T_1 \to \kappa_1 \quad \Delta \vdash_a T_2 \to \kappa_2 \quad \kappa_1, \kappa_2 \leq \mathcal{S}^{\mathbf{l}}}{\Delta \vdash_a T_1; T_2 \to \mathcal{S}^{\max(\kappa_1, \kappa_2)}} \quad \frac{\Delta, x :: \kappa \vdash_a T \to \kappa'}{\Delta \vdash_a \mathbf{rec}\, x :: \kappa\, .\, T \to \kappa'}$$

$$\frac{\Delta \vdash_a T_1 \to \kappa_1 \quad \ldots \quad \Delta \vdash_a T_n \to \kappa_n \quad \kappa_1, \ldots, \kappa_n \leq \mathcal{S}^{\mathbf{l}}}{\Delta \vdash_a \oplus\{l_i{:}T_i\}_{\,1 \leq i \leq n} \to \mathcal{S}^{\mathbf{l}}}$$

$$\frac{\Delta \vdash_a T_1 \to \kappa_1 \quad \ldots \quad \Delta \vdash_a T_n \to \kappa_n \quad \kappa_1, \ldots, \kappa_n \leq \mathcal{S}^{\mathbf{l}}}{\Delta \vdash_a \&\{l_i{:}T_i\}_{\,1 \leq i \leq n} \to \mathcal{S}^{\mathbf{l}}}$$

Figura 3.5: Sistema de *kinding* algorítmico, $\Delta \vdash_a T \to \kappa$

operador de sequenciação $T_1; T_2$ cuja multiplicidade depende dos valores de T_1 e de T_2 (valor máximo entre as duas multiplicidades).

Os tipos funcionais apresentados na figura, são as funções que tanto podem ser lineares como tradicionais ($T_1 \multimap T_2$ ou $T_1 \to T_2$) para os quais as multiplicidades dependem exclusivamente do tipo da função, $\mathbf{l}$ para o primeiro caso (lineares) e $\mathbf{u}$ para o segundo (tradicionais). Os restantes tipos funcionais são os pares (T_1, T_2), com multiplicidade $\mathbf{l}$ e os tipos de dados $[l_i{:}T_i]_{i \in I}$, com multiplicidade igual à multiplicidade máxima dos tipos T_1 até T_n.

Por exemplo, o tipo **!Int** é bem formado e é um tipo de sessão linear ($\mathcal{S}^{\mathbf{l}}$), isto é, só pode ser utilizado uma vez e em tipos para canais. Por outro lado, o tipo **!Int**;**Bool** não é um bem formado porque mistura tipos de sessão com tipos funcionais e a regra do operador de sequenciação determina que os tipos **!Int** e **Bool** têm de ser ambos inferiores ou iguais a $\mathcal{S}^{\mathbf{l}}$, isto é, têm de ser tipos de sessão, qualquer que seja a sua multiplicidade. Esta condição não se verifica porque apesar do género do primeiro tipo ser $\mathcal{S}^{\mathbf{l}}$, o do segundo é $\mathcal{T}^{\mathbf{u}}$ que é um tipo funcional ($\mathcal{T}^{\mathbf{u}} > \mathcal{S}^{\mathbf{l}}$). Por sua vez, o tipo **Skip**;**?Char** é um tipo bem formado e o seu género é $\mathcal{S}^{\mathbf{l}}$. Neste exemplo, ambos os constituintes do operador de sequenciação são tipos de sessão portanto, o género resultante é o que tiver maior multiplicidade, neste caso **?Char**, dado que, $\mathcal{S}^{\mathbf{u}} < \mathcal{S}^{\mathbf{l}}$.

O género dos tipos recursivos **rec** `x :: k . T` depende da associação `x :: k`, ou seja,

da associação de um género κ a uma variável de recursão x. Esta associação pode ser especificada aquando da definição do tipo recursivo (**rec** x :: SL . **!Int**;x) e, nesse caso, é automaticamente introduzida a relação no ambiente de *kinding* Δ. Caso a associação não seja especificada (**rec** x . **!Int**;x) é assumido que o *kind* de x é $\mathcal{S}^{\mathsf{l}}$.

O *kind* do tipo **rec** x :: SU . x é $\mathcal{S}^{\mathsf{u}}$ porque a variável x está no ambiente com o valor $\mathcal{S}^{\mathsf{u}}$. O tipo **rec** x :: SU . **!Int**;x também está bem formado mas o seu género é $\mathcal{S}^{\mathsf{l}}$ porque o género de **!Int** é $\mathcal{S}^{\mathsf{l}}$ que é superior ao valor $\mathcal{S}^{\mathsf{u}}$ (valor da variável x). Por outro lado, o tipo **rec** x . a;x é mal formado se a variável a não estiver no ambiente de *kinding* ou se estiver no ambiente e o seu género indicar que não é um tipo de sessão.

Outro requisito necessário é que o corpo do tipo recursivo (T) seja contrativo. As regras que permitem verificar esta condição estão presentes no artigo de Thiemann e Vasconcelos [20]. [2]

O sistema de *kinding* descrito ao longo desta secção é um sistema algorítmico, isto é, está orientado à sintaxe da linguagem. No entanto, é diferente do sistema apresentado por Thiemann e Vasconcelos [20] que não é algorítmico. Portanto, as provas que asseguram a boa formação dos tipos e que permitem atribuir um determinado *kind* a cada tipo apenas serão válidas para o sistema descrito nesta secção se a seguinte conjetura se verificar.

Dados Δ, T, κ e ρ tais que:

$\Delta \vdash T :: \kappa$ e $\Delta \vdash T \to \rho$,

- $\Delta \vdash T :: \kappa$ então $\Delta \vdash T \to \rho \wedge \quad \kappa \leq \rho$
- $\Delta \vdash T \to \rho$ então $\Delta \vdash T :: \kappa \wedge \quad \kappa \leq \rho$

3.2.2 Equivalência de tipos

O problema de determinar se dois tipos são equivalentes apresenta diversos desafios e está fora do âmbito desta dissertação. No entanto, é importante no processo de verificação da linguagem.

A solução apresentada por Thiemann e Vasconcelos [20] para definir a equivalência entre dois tipos é a bisimulação. Intuitivamente, dois tipos de sessão são equivalentes se tiverem o mesmo comportamento a nível das comunicações que efetuam.

A relação de equivalência ~ (bisimilaridade) é união de todas as bisimulações. Portanto, dois tipos S_1 e S_2 são equivalentes se existir uma bisimulação $\mathcal{R}$ tal que $S_1 \mathcal{R} S_2$. Por exemplo, os tipos $S_1 \triangleq$ **rec** x . +{l:**!Int**, m:y} e $S_2 \triangleq$ **rec** y . +{l:**Skip**, m:y};**!Int** são equivalentes através da relação de bisimulação. O par (S_1, S_2) está obviamente na relação. Ao transitarmos através da etiqueta l obtemos o tipo α ($S_1 \xrightarrow{\oplus l} \alpha$). Do mesmo modo,

[2] A seguinte regra é apresentada no artigo mas não foi implementada porque considerámos não ser necessária

$$\frac{\Delta \vdash_c T_1 \quad \Delta \vdash_c T_2}{\Delta \vdash_c T_1; T_2}$$

se a partir de S_2 transitarmos através da etiqueta l também obtemos α (**Skip**;α mais precisamente). Assim sendo, adicionamos o par (α, α) à relação. Como α se reduz para **Skip**, também se adiciona o par (**Skip**, **Skip**). Ao transitarmos pela etiqueta restante m, obtemos que $S_1 \xrightarrow{\oplus m} S_1$ e $S_2 \xrightarrow{\oplus m} S_2$. A bisimulação é {($S_1$, S_2), (α, α), (**Skip**, **Skip**)}.

Um dos desafios que este problema impõe é o facto do operador de **Skip** e do operador de sequenciação formarem um monoide e como tal devem respeitar as regras que esta condição determina, isto é, o elemento neutro é o operador de **Skip** que representa a identidade do operador de sequenciação quer à esquerda **Skip**;**!Int** ou à direita **!Int**;**Skip**. Outra dificuldade é a necessidade do operador recursivo **rec** x . **!Int**;x ser equivalente ao seu desenvolvimento **!Int**;(**rec** x . **!Int**;x).

Como descrito por Thiemann e Vasconcelos [20], é possível traduzir um tipo de sessão num processo BPA (*basic process algebra*) de tal modo que a noção de equivalência coincide com a noção de bissimilaridade em processos BPA.

O artigo de Christensen et al. [6] mostra que a noção de bissimilaridade de processos BPA é decidível, contudo não define diretamente um algoritmo. Em vez disso, tentou-se um algoritmo baseado nas ideias apresentadas por Jančar e Moller [15] que traduz um tipo de sessão numa gramática independente do contexto a partir da qual se gera uma árvore de expansão.

Contudo, o algoritmo implementado não é completo, visto que existem tipos que sabemos serem equivalentes e que o algoritmo diz não o serem.

3.2.3 Verificação de tipos

A verificação de tipos visa determinar se uma expressão tem o tipo esperado. A verificação é baseada num sistema bidirecional, ou seja, um sistema de distingue duas relações: a relação de síntese de um tipo e a relação de verificação de um tipo de encontro a outro.

Síntese Dado um contexto de variáveis de tipo Δ, um contexto de variáveis de programa Γ e uma expressão e, sintetizar o tipo T:

$$\frac{\overbrace{\ldots}^{\text{Específico para cada expressão e}}}{\underbrace{\Delta;\Gamma \vdash \mathsf{e}}_{\text{In}} \rightarrow \underbrace{T;\Gamma}_{\text{Out}}}$$

Verificação Dados os contexto Δ e Γ, uma expressão e e um tipo esperado T, verificar se o tipo da expressão U é equivalente ao tipo esperado (através da relação de equivalência ~):

$$\frac{\Delta;\Gamma_1 \vdash U;\Gamma_2 \qquad \Delta \vdash U \sim T}{\underbrace{\Delta;\Gamma_1 \vdash e : T}_{\text{In}} \rightarrow \underbrace{\Gamma_2}_{\text{Out}}}$$

A verificação de tipos é apresentada em quatro categorias distintas: a verificação de expressões, a verificação das declarações de tipos de dados, a verificação de funções e a verificação de programas.

Verificação de expressões

De seguida são apresentadas diversas figuras que contêm as regras necessárias para a verificação de tipos que é feita através da avaliação da função $\Delta; \Gamma \vdash_a e \to T$.

A figura 3.6 apresenta uma única regra que é genérica para todos os tipos básicos (**Int**, **Bool**, **Char** e ()) e indica que para um valor `c` é retornado o seu tipo, sem qualquer verificação adicional. Por exemplo, a verificação da expressão `2` é feita pela avaliação de $\Delta; \Gamma \vdash_a 2 \to T$ e retorna o tipo **Int** (porque o valor `2` é um inteiro).

Expressões básicas

$$\frac{\mathbf{typeof}\ (c) = T}{\Delta; \Gamma \vdash_a c \to T; \Gamma}$$

Figura 3.6: Verificação de tipos algorítmica para expressões básicas

A figura 3.7 apresenta duas regras que permitem verificar variáveis e expressões **let**.

Para verificar uma variável x é feita a avaliação da função $\Delta; \Gamma \vdash_a x \to T$. Ambas as regras verificam se a variável `x` se encontra no ambiente de variáveis ($x : T \in \Gamma$). Caso se encontre, é verificada a multiplicidade (`lin` ou `un`) do tipo T que lhe está associado no ambiente Γ. Se o tipo for partilhado (**u**) é devolvido o tipo T e o ambiente de variáveis Γ que contem `x`. Caso contrário, se o tipo for linear (**l**), também é devolvido o tipo T e ambiente Γ mas sem a variável `x` porque esta só pode ser utilizada uma vez.

A verificação de uma expressão **let** `x = e1` **in** `e2` consiste em avaliar as duas expressões que a compõem (`e1` e `e2`) sendo que, o valor final que é devolvido é o valor da segunda expressão. O valor da primeira expressão T_1, é adicionado ao ambiente Γ associado à variável `x`. Por exemplo, a expressão **let** `x =` **True in** `2` adiciona a associação `x :` **True** ao ambiente e termina a avaliação com o tipo **Int** (valor resultante da avaliação da segunda expressão).

A regra que permite avaliar a aplicação de expressões é $\Delta; \Gamma \vdash_a e_1\, e_2 \to U_2; \Gamma_3$. Segundo a regra disponível na figura 3.8, primeiro avalia-se a primeira expressão e_1 da qual resulta do tipo T e o ambiente Γ_2. De seguida, é utilizada uma função de extração ($\rightsquigarrow$) que representa a verificação e extração de um tipo. Neste caso em concreto, a operação $T \rightsquigarrow_{(\to|\multimap)} U_1 \to_m U_2$ extrai e verifica se existem uma função (linear $\multimap$ ou partilhada $\to$) no tipo T. Por exemplo, se o tipo T for **Int** $\to$ **Bool** a função de extração devolve os tipos **Int** e

Variáveis

$$\frac{\Gamma(x) = T \quad \Delta \vdash_a \mathsf{lin}\,(T)}{\Delta;\Gamma \vdash_a x \to T;\Gamma \setminus \{x\}} \qquad \frac{\Gamma(x) = T \quad \Delta \vdash_a \mathsf{un}\,(T)}{\Delta;\Gamma \vdash_a x \to T;\Gamma}$$

Let

$$\frac{\Delta;\Gamma \vdash_a e_1 \to T_1;\Gamma_2 \quad \Delta;\Gamma, x : T_1 \vdash_a e_2 \to T_2;\Gamma_3}{\Delta;\Gamma_1 \vdash_a \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \to T_2;\Gamma_3}$$

Figura 3.7: Verificação de tipos algorítmica para varáveis e Let

Bool. No caso de T ser apenas **Int** a verificação da aplicação falha porque é necessário que T seja uma função e termina com um erro. De seguida, é verificada a segunda expressão da aplicação (e_2) de encontro ao primeiro tipo (U_1) extraído. Por fim, a avaliação termina devolvendo o segundo tipo que resultou da extração (U_2) e o ambiente (Γ_3).

Uma aplicação de tipos $e\,[T]$, necessita que o tipo da expressão e seja um esquema de tipos (**forall** a :: k ⇒ U) para que seja possível substituir em U todas as ocorrências de a por T. A função de extração ($\rightsquigarrow_\forall$) verifica se o tipo da expressão e é um esquema de tipos e extrai as associações de variáveis a *kinds* (a :: k) do esquema de tipos e também o tipo U que o constitui. Por último, é verificado se o tipo $U\,[T/\alpha]$ (tipo substituído) é bem formado, isto é, se o *kinding* apresentado em 3.2.1 não apresenta qualquer erro. A avaliação termina com o valor do tipo substituído $U\,[T/\alpha]$.

Aplicação

$$\frac{\Delta;\Gamma_1 \vdash_a e_1 \to T;\Gamma_2 \quad T \rightsquigarrow_{(\to|\multimap)} (U_1, U_2) \quad \Delta;\Gamma_2 \vdash_a e_2 \leftarrow U_1;\Gamma_3}{\Delta;\Gamma_1 \vdash_a e_1\, e_2 \to U_2;\Gamma_3}$$

Aplicação de tipos

$$\frac{\Delta;\Gamma_1 \vdash_a e \to T_1;\Gamma_2 \quad T_1 \rightsquigarrow_\forall ([\alpha :: \kappa], U) \quad \Delta \vdash_a T \leftarrow \kappa}{\Delta;\Gamma_1 \vdash_a e\,[T] \to U\,[T/\alpha];\Gamma_2}$$

Figura 3.8: Verificação de tipos algorítmica para aplicações

A figura 3.9 apresenta a regra que permite verificar uma expressão condicional tradicional (**if** e1 **then** e2 **else** e3). A primeira expressão (e1) é verificada de encontro ao tipo **Bool**. De seguida, é verificado o tipo da expressão e2 que, posteriormente, é utilizado para verificar a expressão e_3 (têm de ser iguais). Finalmente, verifica-se a equivalência dos

ambientes finais (resultantes de e2 e e3) (através da relação de equivalência de contextos ~).

Condicional

$$\frac{\Delta;\Gamma_1 \vdash_a e_1 \leftarrow \mathbf{Bool};\Gamma_2 \quad \Delta;\Gamma_2 \vdash_a e_2 \rightarrow T_1;\Gamma_3 \quad \Delta;\Gamma_2 \vdash_a e_3 : T_2;\Gamma_4 \leftarrow T_1 \quad \Delta \vdash_a \Gamma_3 \sim \Gamma_4}{\Delta;\Gamma_1 \vdash_a \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rightarrow T_1;\Gamma_4}$$

Figura 3.9: Verificação de tipos algorítmica para a expressão condicional

As regras que permitem verificar as operações sobre pares estão presentes na figura 3.10. É composta por uma regra para o construtor de pares (e_1, e_2) e outra para o destrutor de pares **let** x,y = e1 **in** e2. A primeira verifica o tipo de cada uma das expressões e_1 e e_2 e termina devolvendo um par com os tipos resultantes de cada expressão (T_1, T_2). A regra para o destrutor, começa por avaliar a primeira expressão e_1 e com base no tipo resultante T verifica-se se T é composto por um par de valores e extrai-se esse par. Os tipos que constituem o par extraído (U_1, U_2) são associados a cada uma das variáveis (x e y) e são colocados no ambiente de variáveis Γ_2. A última verificação é sobre a expressão e_2 cujo resultado é o tipo resultante da expressão **let** x,y = e1 **in** e2.

Construtor de pares

$$\frac{\Delta;\Gamma_1 \vdash_a e_1 \rightarrow T_1;\Gamma_2 \quad \Delta;\Gamma_2 \vdash_a e_2 \rightarrow T_2;\Gamma_3}{\Delta;\Gamma_1 \vdash_a (e_1, e_2) \rightarrow (T_1, T_2);\Gamma_3}$$

Destrutor de pares

$$\frac{\Delta;\Gamma_1 \vdash_a e_1 : T;\Gamma_2 \quad T \rightsquigarrow_{\otimes} (U_1, U_2) \quad \Delta;\Gamma_2, x : U_1, y : U_2 \vdash_a e_2 \rightarrow U;\Gamma_3}{\Delta;\Gamma_1 \vdash_a \mathbf{let}\ x, y = e_1\ \mathbf{in}\ e_2 \rightarrow U;\Gamma_3}$$

Figura 3.10: Verificação de tipos algorítmica para operações sobre pares

A figura 3.11 é composta por uma regra para cada uma das operações de comunicação (**new**, **send**, **receive**, **select** e **match**).

A regra para a operação de criação de canais (**new** T) verifica se T é um tipo de sessão linear ($\mathcal{S}^1$) e termina devolvendo um par com o tipo T e o seu dual $(T, \overline{T})$.

A operação **send** $e_1\ e_2$ é verificada em quatro partes. A primeira consiste em verificar as duas expressões e_1 e e_2 separadamente. De seguida, confere-se o tipo de e_1 através da função de extração $(T_1 \rightsquigarrow_B B_1)$ que verifica que T_1 é um tipo básico. Na terceira parte

Criação de canais

$$\frac{\Delta \vdash_a T \leftarrow \mathcal{S}^{\mathsf{l}}}{\Delta;\Gamma \vdash_a \mathbf{new}\ T \to (T,\overline{T});\Gamma}$$

Envio de valores em canais

$$\frac{\Delta;\Gamma_1 \vdash_a e_1 \to T_1;\Gamma_2 \quad \Delta;\Gamma_2 \vdash_a e_2 \to T_2;\Gamma_2 \quad T_1 \rightsquigarrow_B B_1 \quad T_2 \rightsquigarrow_! (B_2,U) \quad B_1 = B_2}{\Delta;\Gamma_1 \vdash_a \mathbf{send}\ e_1\ e_2 \to U;\Gamma_2}$$

Receção de valores em canais

$$\frac{\Delta;\Gamma_1 \vdash_a e_1 \to T;\Gamma_2 \quad T \rightsquigarrow_? (B,U)}{\Delta;\Gamma_1 \vdash_a \mathbf{receive}\ e \to (B,U);\Gamma_2}$$

Seleção de etiquetas

$$\frac{\Delta;\Gamma \vdash_a e \to T;\Gamma_2 \quad T \rightsquigarrow_\oplus C:U}{\Delta;\Gamma \vdash_a \mathbf{select}\ C\ e \to U;\Gamma_2}$$

Ramificação

$$\frac{\begin{array}{l}\Delta;\Gamma_1 \vdash_a e \to T;\Gamma_2 \\ T \rightsquigarrow_\& C_1:T_1 \\ \Delta;\Gamma_2,x:T_1 \vdash_a e_1 \to U;\Gamma_1'\end{array} \quad \left(\begin{array}{c} T \rightsquigarrow_\& C_i:T_i \\ \Delta;\Gamma_2,x:T_i \vdash_a e_i:U_i;\Gamma_i \leftarrow U \\ \Delta \vdash_a \Gamma_1' \sim \Gamma_i' \end{array}\right)^{i>1}}{\Delta;\Gamma_1 \vdash_a \mathbf{match}\ e\ \mathbf{with}\ \{C_i x \to e_i\} \to U;\Gamma_1}$$

Figura 3.11: Verificação de tipos algorítmica para as operações de comunicação

avalia-se se o tipo do canal de comunicação (e_2) é composto por um tipo **!**B através da função de extração ($T_2 \rightsquigarrow_! (B,U)$). Esta por sua vez, faz as verificações necessárias e devolve um par onde o primeiro elemento é o tipo básico da operação de envio (B) e o segundo é a continuação U (no caso do tipo ser **!**B;U). Por último, testa-se se os tipos básicos B_1 e B_2 são iguais (e portanto equivalentes).

Para verificar a expressão **receive** e verifica-se se o tipo de e corresponde a uma receção de um tipo básico (?B). A operação $T \rightsquigarrow_? (B,U)$, extrai de T um par composto pelo tipo básico da operação de receção e o tipo restante. Por exemplo, se o tipo T for ?**Int**;**!Bool**;?**Char** a operação $\rightsquigarrow_?$ devolve o par (?**Int**, **!Bool**;?**Char**).

Na verificação da expressão **select** e garante-se que o tipo T é composto por uma escolha externa ($\oplus$) para que seja possível, caso seja composto, extrair o construtor C cujo tipo associado (U) é devolvido.

Para a expressão **match** e **{**$C_i x \to e_i$**}** é expectável que o tipo de e seja uma escolha interna ($\&$) a partir da qual se extrai o primeiro construtor e o seu respetivo tipo. Depois

de adicionar a associação $x : T_1$ ao ambiente Γ e de extrair o tipo U da primeira expressão e_1 repete-se o mesmo processo para os restantes construtores com a diferença que as expressões são verificadas de encontro ao tipo U. A avaliação termina com o tipo U.

A figura 3.12 apresenta a regra para verificar a expressão **fork** e que necessita que o tipo T da expressão e seja um tipo partilhado, isto é, que tenha uma multiplicidade do tipo $\mathbf{u}$. No fim, a avaliação desta expressão retorna o tipo unitário $()$.

Criação de fios de execução

$$\frac{\Delta; \Gamma_1 \vdash_a e \to T; \Gamma_2 \quad \Delta \vdash_a \text{un}(T)}{\Delta; \Gamma_1 \vdash_a \mathbf{fork}\ e \to (); \Gamma_2}$$

Figura 3.12: Verificação de tipos algorítmica para a expressão fork

Na figura 3.13 são apresentadas regras que permitem validar os construtores C e o destrutor de tipos de dados de **case** e **of {**$C_i x_i : e_i$**}**. Na validação dos construtores apenas é verificado se estes existem no ambiente Γ. A verificação das expressões **case** é análoga à das expressões **match**, com a diferença que nesta avaliação podem existir diversos argumentos para cada construtor.

Construtor

$$\frac{\Gamma(C) = T}{\Delta; \Gamma \vdash_a C \to T; \Gamma}$$

Destrutor de tipos de dados

$$\frac{\begin{array}{l}\Delta; \Gamma_1 \vdash_a e \to T; \Gamma_2 \\ T \rightsquigarrow_{[1]} C_1 : T_1 \\ \Delta; \Gamma_2, x : T_1 \vdash_a e_1 \to U; \Gamma_1'\end{array} \quad \left(\begin{array}{c} T \rightsquigarrow_{[i]} C_i : T_i \\ \Delta; \Gamma_2, x : T_i \vdash_a e_1 : U_i \to \Theta_i \\ \Delta \vdash_a \Gamma_1' \sim \Theta_i' \end{array}\right)^{i>1}}{\Delta; \Gamma_1 \vdash_a \mathbf{case}\ e\ \mathbf{of}\ \{C_i x : e_i\} \to U; \Gamma_1}$$

Figura 3.13: Verificação de tipos algorítmica para expressões sobre tipos de dados

Verificação de declarações de tipos de dados

As regras (figura 3.14) para uma declaração de um tipo de dados **data** $\alpha = x\, T_1 \ldots T_n$ impõem que o *kind* κ seja um tipo partilhado, ou seja, é necessário que a relação $k \geq \mathcal{T}^{\mathbf{u}}$ se verifique. Por outro lado, é também necessário que todos os tipos T_{i_j} do tipo de dados (DD) sejam bem formados (representado por $\Delta \vdash_a T_{ij} \to \kappa_{ij}$).

$$\text{DD} \ ::= \ \varepsilon \ \mid \ \text{DD } \textbf{data } \alpha = C\, T_{i_1} \ldots T_{m_n}, \quad n \geq 0$$

$$\boxed{\Delta \vdash_a \text{DD} \to \Delta} \qquad \frac{}{\Delta \vdash_a \varepsilon \to \varepsilon} \qquad \frac{\Delta_1 \vdash_a \text{DD} \to \Delta_2 \qquad \Delta_2 \vdash_a T_{i_j} :: \kappa_{i_j}}{\Delta_1 \vdash_a \text{DD } \textbf{data } \alpha = (C T_{1_i} \ldots T_{m_n}) \to \Delta_2, \alpha :: \mathcal{T}^{\mathbf{u}}}$$

Figura 3.14: Verificação de tipos para as declarações dos tipos de dados (DD)

Verificação de funções

A verificação de funções está dividida em duas partes essenciais. A primeira parte verifica as declarações dos tipos das funções (assinaturas) enquanto que a segunda verifica as declarações das funções, o corpo da função.

Para verificar as assinaturas das funções $x :: \mathcal{C}$ (figura 3.15) é necessário verificar se o tipo $\mathcal{C}$ da função x é bem formado ($\Delta \vdash_a \mathcal{C} :: \kappa$).

$$\text{SD} \ ::= \ \varepsilon \ \mid \ \text{SD } x :: \mathcal{C}$$

$$\boxed{\Delta, \Gamma \vdash_a \text{SD} \to \Gamma} \qquad \frac{}{\Delta, \Gamma \vdash_a \varepsilon \to \Gamma} \qquad \frac{\Delta, \Gamma_1 \vdash_a \text{SD} \qquad \Delta \vdash_a \mathcal{C} \to \kappa}{\Delta, \Gamma_1 \vdash_a \text{SD} \quad x :: \mathcal{C} \to \Delta, x :: \mathcal{C}}$$

Figura 3.15: Verificação de tipos para as declarações de tipos de funções (SD)

A figura 3.16 é composta por uma regra que permite verificar as declarações de funções (corpo da função). Para isso, o tipo da função é decomposto em subtipos com base nas funções. Por exemplo, o tipo **Int** → **Char** → **Bool** é decomposto em três parcelas **Int**, **Char** e **Bool** respetivamente. Assim sendo, o tipo fica decomposto em $T_1 \ldots T_{n+1}$ o que permite associar os tipos T_1 ao primeiro argumento, o tipo T_2 ao segundo argumento e o tipo T_{n+1} ao tipo de retorno da função que deve ser equivalente ao tipo obtido através da avaliação da expressão e (secção 3.2.3). Também é verificado se todos os tipos presentes no ambiente Γ são tipos tradicionais (multiplicidade $\mathbf{u}$).

Verificação de programas

A verificação de um programa é a verificação principal e mais genérica que utiliza todas as verificações supra descritas. Primeiro são verificadas as declarações de tipos de dados, de seguida as declarações de tipos de funções e finalmente as declarações de funções (que implica verificar todas as expressões do programa P). Este sistema tem um ambiente de variáveis inicial representado por Γ_0 na figura 3.17 que é correspondente ao *prelude* da

$$\text{FD} \ ::= \ \varepsilon \ \mid \ \text{FD}\ x\ x_1\ ...\ x_n\ =\ e$$

$\boxed{\Delta;\Gamma \vdash_a \text{FD}}$ $\qquad \dfrac{}{\Delta;\Gamma \vdash_a \varepsilon}$

$$\frac{\begin{array}{cc} \Delta;\Gamma_1 \vdash_a \text{FD} & \Delta;\Gamma_1 \vdash_a \Gamma_1(x) = \alpha :: \kappa \Rightarrow T_1 \to ... \to T_{n+1};\Gamma_2 \\ \Delta, \alpha :: \kappa \ ; \ \Gamma, x_1 : T_1\ ...\ x_n : T_n \vdash_a e \to U;\Gamma_3 & \mathsf{un}\,(\Gamma_3) \qquad \Delta \vdash_a T_{n+1} \sim U \end{array}}{\Delta;\Gamma_1 \vdash_a \text{FD}\ \ x\ x_1\ ...\ x_n\ =\ e \to U,\Gamma_3}$$

Figura 3.16: Verificação de tipos para as declarações de funções (FD)

linguagem Haskell. Considera-se portanto que o *prelude* é o estado inicial deste ambiente.

$$\text{P} \ ::= \ \text{DD}\ \ \text{SD}\ \ \text{FD} \qquad \frac{\vdash_a \text{DD} \to \Delta \quad \Delta, \Gamma_0 \vdash_a \text{SD} \to \Gamma \quad \Delta \vdash_a \text{FD}}{\Delta;\Gamma_0 \vdash_a \text{DD}\ \ \text{SD}\ \ \text{FD}}$$

Figura 3.17: Verificação de tipos para os programas (P)

O sistema que foi apresentado nesta secção é algorítmico contrariamente ao sistema que é apresentado por Thiemann e Vasconcelos [20]. Deste modo, as provas que sustentam o sistema apresentado no artigo apenas serão válidas para o sistema implementado se a conjetura apresentada abaixo for verdadeira. Não foram feitas provas que comprovem o valor de verdade destas relações. No entanto, através da implementação de diversos exemplos significativos, nomeadamente os presentes no artigo, constatámos que o sistema de verificação de tipos apresentado teve o comportamento esperado.

Dados Δ, Γ, T e e,

$\Delta \vdash T :: \kappa$ e $\Delta \qquad \vdash T \to \rho$,

- Se $\Delta;\Gamma \vdash e : T \quad$ então $\quad \Delta;\Gamma \vdash e \to U \ \wedge \ T <: U$
- Se $\Delta;\Gamma \vdash e \to T \quad$ então $\quad \Delta;\Gamma \vdash e : T$

Capítulo 4

Implementação

Para implementar a linguagem recorremos à linguagem Haskell em dois sentidos: o compilador foi escrito nesta linguagem e é também a linguagem alvo do compilador, isto é, geramos código Haskell. Este capítulo começa com uma descrição das fases do compilador (secção 4.1) onde é detalhada a forma como o compilador foi construido. Inclui-se uma descrição de cada um dos seus componentes. De seguida são descritos os testes a que o compilador foi sujeito (secção 4.2) e os resultados obtidos.

4.1 Fases do compilador

Esta secção apresenta um esquema genérico do compilador (figura 4.1) que ilustra a forma como foi implementado. A primeira fase do compilador é composta pelo *lexer* e pelo *parser*, descritos na secção 4.1.1, que recebem um ficheiro FreeST (*.cfs*) e o transformam numa árvore sintática. Na secção 4.1.2 são apresentados detalhes da implementação dos sistema de validação da linguagem anteriormente descrito. A secção 4.1.3 descreve o processo de geração de código implementado.

A última parte representada na figura 4.1 é referente ao compilador da linguagem Haskell (GHC) que é relevante uma vez que o resultado do processo de geração de código é um programa Haskell que é, posteriormente compilado com este compilador originando um ficheiro que contém um ficheiro com o programa compilado.

4.1.1 Lexer e Parser

Os analisadores lexicográfico (*lexer*) e sintático (*parser*) da linguagem FreeST foram implementados através do *Parsec*, um combinador de analisadores. Estes combinadores são uma técnica para expressar analisadores recursivos descendentes. Recebem o *input* como parâmetro e procuram reconhecer os seus caracteres. Quando a análise feita por um analisador sintático (*parser*) é bem sucedida, é retornado um resultado (representação interna da linguagem) e o resto da sequência de valores de entrada (*input*) para que esta

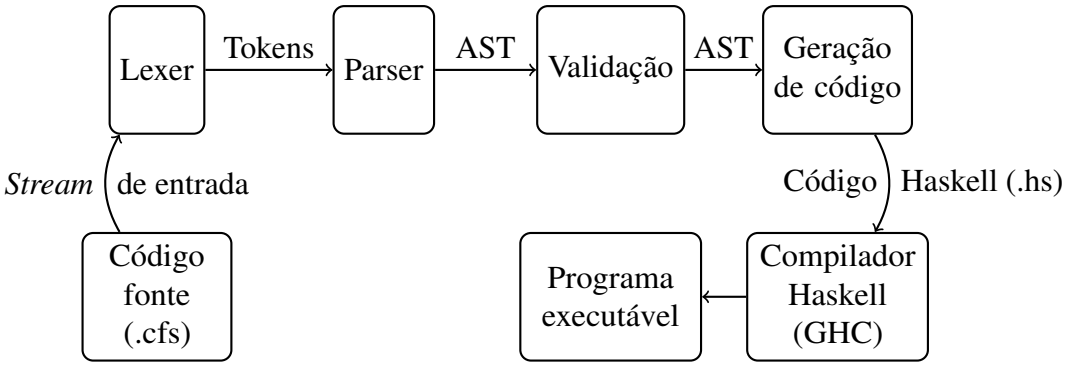


Figura 4.1: Fases do compilador

possa continuar a ser avaliada. Quando não é possível reconhecer um dos caracteres do *input* é lançado um erro.

A objetivo da utilização destes combinadores é construir *parsers* mais complexos através da composição de *parsers* mais simples, ao invés da composição sintática de um analisador recursivo como é tradicional.

De seguida são apresentados exemplos de combinadores que foram implementados. O primeiro, parseVar (linhas 1 – 4) reconhece uma variável, isto é, um identificador (linha 3) e retorna a sua representação interna. O segundo, parsePair (linhas 6 – 11) reconhece um par de tipos separados por uma vírgula recorrendo ao analisador parseType que reconhece um tipo da linguagem. O *parser* parseRec reconhece um tipo recursivo na forma **rec** x :: k . T. Inicialmente reconhece o lexema **rec** (linha 15) seguido de um identificador (linha 16). O parser option presente na linha 17 tenta aplicar o *parser* parseVarBind que reconhece a sequência composta por " :: " e um dos géneros: SL, SU, TL, TU. Se este falhar, devolve o valor por defeito Kind Session Lin. Se for bem sucedido devolve o valor obtido. De seguida reconhece a sequência comporta por um ponto (dot) e um tipo (parseType). Por fim, retorna a representação interna de um tipo recursivo.

```
1  parseVar :: Parsec String u Type
2  parseVar = do
3    id <- identifier
4    return (Var id)
5
6  parsePair :: Parsec String u Type
7  parsePair = parens do
8    t <- parseType
9    comma
10   u <- parseType
11   return (PairType t u)
12
13 parseRec :: Parsec String u Type
14 parseRec = do
```

```
15    rec
16    id <- identifier
17    k <- option (Kind Session Lin) parseVarBind
18    dot
19    t <- parseType
20    return (Rec (Bind id k) t)
```

Árvore sintática

O *parser* da linguagem é responsável por traduzir a sequência de caracteres que é fornecida como *input* na representação interna da linguagem. A representação interna das expressões é armazenada originando uma árvore sintática. Durante este processo, é utilizado um mónade de estado que permite guardar o ambiente de variáveis, o ambiente de expressões, o ambiente de construtores e ainda o ambiente de *kinding*. Assim sendo, o resultado do *parser* é o tuplo:

```
(VarEnv, ExpEnv, ConsEnv, KindEnv)
```

Ambiente de variáveis

O ambiente de variáveis tem o tipo:

```
type VarEnv = Data.Map.Strict String Type
```

É composto pelas assinaturas das funções, isto é, o nome da função e o seu tipo. Por exemplo, para a função com a assinatura fun :: **Int** → **Bool** é adicionada a entrada "fun" ↦ (**Int** → **Bool**).

Ambiente de expressões

O ambiente de expressões tem o tipo:

```
type ExpEnv = Data.Map.Strict String ([String], Expression)
```

É composto pelas expressões associadas à sua função. Por exemplo, uma função fun x y z = e adiciona ao ambiente a entrada que mapeia a **String** "fun" para um par composto pela lista de argumentos ["x","y","z"] e a expressão e. Assim sendo, é adicionada a entrada "fun" ↦ (["x","y","z"], e).

Ambiente de construtores

O ambiente de construtores tem o tipo:

```
type ConsEnv = Data.Map.Strict String TypeScheme
```

É composto por todos os componentes dos tipos de dados. Por exemplo, o tipo de dados **data** Tree = Leaf | Node **Int** Tree Tree é decomposto em duas funções: Leaf :: Tree e Node :: **Int** → Tree → Tree → Tree que são também são adicionadas ao ambiente.

Ambiente de géneros

O ambiente de *kinding* tem como tipo:

```
type KindEnv = Data.Map.Strict String Kind
```

Como descrito anteriormente, o ambiente é composto pelas associações entre todas as variáveis e o seu género (atribuído pelo sistema de *kinding* descrito na secção 3.2.1).

Para o exemplo do tipo de dados definido acima (Tree) é adicionada ao ambiente de géneros a entrada "Tree" ↦ (Kind Functional Un) uma vez que este é o género por omissão para os tipos de dados.

Restrições sintáticas

Durante a fase de *parsing* é necessário verificar se existem duplicados. Isto é, se existem funções, declarações de tipos de funções, declarações de tipos de dados ou ainda definições de construtores declarados mais que uma vez. Por exemplo, não é possível declarar mais que uma vez o tipo de dados **data** Tree mesmo que sejam definições diferentes (compostos por construtores diferentes). Por outro lado, não é possível definir tipos de dados que ainda que sejam diferentes como por exemplo **data** Tree1 e **data** Tree2 tenham nas suas definições o mesmo construtor (**data** Tree1 = Leaf e **data** Tree2 = Leaf).

As verificações acima descritas são feitas na fase de *parsing* e não na fase de verificação, dado que os ambientes são mapas (Data.Map.Strict) e como tal não é possível que existam duas chaves iguais, impossibilitando assim que ambas as definições estejam no ambiente para serem verificadas na fase de validação (onde são feitas todas as restantes validações).

4.1.2 Verificação de tipos

A fase de validação da linguagem foi apresentada na secção 3.2 que descreve as regras necessárias para garantir que um programa está bem formado. Para que seja possível verificar todas as expressões, funções, assinaturas e declarações de tipos de dados é necessário percorrer os diferentes ambientes para garantir que todos os componentes da linguagem são testados.

Ao longo de toda a fase de validação é utilizada uma mónade de estado que permite manter o estado do compilador e é composto pelo tuplo de ambientes (VarEnv, ExpEnv, ConsEnv, KindEnv) e por uma lista de erros que acumula todos os erros encontrados ao longo da verificação de modo a fornecer ao utilizador informação mais precisa acerca dos erros do programa.

A figura 3.17 mostra a ordem pela qual um programa é verificado. A primeira fase consiste em verificar todas as declarações de tipos de dados que estão no ambiente ConsEnv. Por exemplo, a declaração de um tipo de dados:

```
data Tree = Leaf | Node Int Tree Tree
```

é adicionada ao ambiente com duas entradas distintas, uma por cada um dos construtores: "Node" ↦ (**Int** → Tree → Tree → Tree) e "Leaf" ↦ Tree. Nestes casos, é verificado se os tipos **Int** → Tree → Tree → Tree e "Leaf" são bem formados. Na segunda fase são verificados todos os tipos das funções para garantir que se encontram bem formados. A terceira e última fase da verificação avalia os corpos das funções. Isto é, determina se os tipos das expressões correspondem aos tipos que foram declarados como tipos das funções.

A ordem pelas qual são feitas as verificações permite tratar as declarações mutuamente recursivas. Por exemplo, consideremos as seguintes as seguintes funções:

```
f :: Int → Int
f x = g x

g :: Int → Int
g x = f x
```

Neste caso, na primeira fase as assinaturas das funções f e g são colocadas no ambiente. Na segunda fase, verificam-se as assinaturas das duas funções f (**Int** → **Int**) e g (**Int** → **Int**) que estão no ambiente VarEnv para determinar se os tipos estão bem formados segundo as regras apresentadas na secção 3.2.1. Ainda nesta fase, analisam-se as expressões das funções f e g para determinar se estas têm o tipo esperado (**Int** em ambos os casos). Ao analisar o corpo da função f é necessário saber o tipo da expressão que o compõe, ou seja, o tipo da função g que já está no ambiente. A verificação da função g é análoga.

4.1.3 Geração de código

A linguagem alvo do compilador é Haskell e como tal é necessário implementar um processo que seja capaz de traduzir a linguagem FreeST para a linguagem Haskell. Deste modo, é possível utilizar o compilador *GHC* para compilar o código gerado. O processo de tradução que foi implementado apresenta quatro desafios principais:

- A linguagem que apresentamos é *call-by-value* e o Haskell (linguagem alvo da geração de código) é *call-by-name*, isto é, apenas computa as expressões que são passadas como argumento quando estas são utilizadas. Para resolver esta questão, tirámos partido da extensão *BangPatterns* que a linguagem Haskell disponibiliza e que permite forçar a avaliação das expressões que são passadas como argumento. O processo de tradução apenas necessita de garantir que é utilizado um ponto de exclamação antes de cada parâmetro das funções. Uma função `fun x = e` quando traduzida fica `fun !x = e`. É também necessário que o ficheiro traduzido contenha a indicação da extensão da linguagem `{-# LANGUAGE BangPatterns #-}` explícita.

- O código gerado relativo às operações de comunicação presentes na figura 3.3 foi implementado recorrendo a duas *MVar* por canal (uma para cada extremidade do

canal). Uma *MVar* é uma célula de memória mutável que tem apenas dois estados, ou está vazia ou tem com um valor.

As *MVar* têm três operações fundamentais. A operação putMVar permite escrever na zona de memória partilhada: corresponde à operação de **send**. A operação takeMVar lê da zona da *MVar*: corresponde à operação **receive**. E a operação newEmptyMVar permite criar novas *MVar* e implementa a operação de **new** da nossa linguagem.

Na listagem 4.1 são apresentadas as implementações das operações **send**, **receive** e **new** que possibilitam a comunicação em canais. É ainda possível observar nesta listagem que a implementação da operação **new** cria duas *MVar*, isto deve-se ao facto dos canais da linguagem serem bidirecionais e ser necessária uma *MVar* para codificar cada uma das extremidades de um canal.

- Em Haskell uma *MVar* t têm um tipo associado (t) que se mantém inalterado durante o curso da computação. Como esta é utilizada para a implementação dos canais de comunicação necessitamos que o seu tipo possa variar de modo a tornar possível o envio de, por exemplo, um valor inteiro seguido de um booleano (!**Int** ;!**Bool**). Para que o sistema de tipos do Haskell não verifique os tipos que circulam nos canais utilizámos uma primitiva que converte um valor de um tipo noutro. Apesar de insegura esta primitiva não apresenta qualquer problema porque fazemos a nossa própria verificação de tipos. As funções que concretizam o envio e a receção de valores em canais, mostram o uso desta primitiva (unsafeCoerce) e ainda das *MVar* cuja criação está representada pela função _new:

```
1  _new :: IO ((MVar a, MVar b), (MVar b, MVar a))
2  _new = do
3    m1 <- newEmptyMVar
4    m2 <- newEmptyMVar
5    return ((m1, m2), (m2, m1))
6
7  _send :: a → (MVar b, MVar c) → IO (MVar b, MVar c)
8  _send x (m1, m2) = do
9    putMVar m2 (unsafeCoerce x)
10   return (m1, m2)
11
12 _receive :: (MVar a, MVar b) → IO (c, (MVar a, MVar b))
13 _receive (m1, m2) = do
14   a <- takeMVar m1
15   return (unsafeCoerce a, (m1, m2))
```

Listing 4.1: Operações sobre canais

- Como a linguagem alvo do compilador é Haskell, as operações de envio (**send** v c),

de receção ($\mathbf{receive}\ c$), de criação de canais ($\mathbf{new}\ T$) e de fios de execução ($\mathbf{fork}\ e$) são forçosamente operações sobre um mónade, mais precisamente, são operações de IO.

As operações acima descritas representam uma dificuldade: decidir quando traduzir as expressões para código de um mónade ou não. Para resolver este problema, antes de aplicar a função de tradução, anotamos a árvore sintática com valores booleanos que representam o estado que é esperado de cada expressão, isto é, se é expectável que uma dada expressão seja uma operação sobre um mónade. As regras que foram implementadas para anotar a árvore sintática encontram-se na figura 4.2.

$$\frac{}{E \vdash () \to \mathbf{False}} \quad \frac{}{E \vdash x \to \mathbf{False}} \quad \frac{}{E \vdash c \to \mathbf{False}} \quad \frac{}{E \vdash b \to \mathbf{False}}$$

$$\frac{}{E \vdash x \to \mathbf{False}} \quad \frac{E \vdash e_1 \to b_1 \quad E, x : b_1 \vdash e_2 \to b_2}{E \vdash \mathbf{let}\ x = e1\ \mathbf{in}\ e2 \to b_2}$$

$$\frac{E \vdash e_1 \to b_1 \quad E \vdash e_2 \to b_2}{E \vdash e_1 e_2 \to \mathbf{False}} \quad \frac{E \vdash e \to b_1}{E \vdash e\,[T] \to \mathbf{False}}$$

$$\frac{E \vdash e_1 \to b_1 \quad E \vdash e_2 \to b_2 \quad E \vdash e_3 \to b_3}{E \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \to b_3}$$

$$\frac{E \vdash e_1 \to b_1 \quad E \vdash e_2 \to b_2}{E \vdash (e_1, e_2) \to \mathbf{False}} \quad \frac{E \vdash e_1 \to b_1 \quad E \vdash e_2 \to b_2}{E \vdash \mathbf{let}\ x, y = e_1\ \mathbf{in}\ e_2}$$

$$\frac{}{E \vdash \mathbf{new}\ T} \quad \frac{E \vdash e_1 \to b_1 \quad E \vdash e_2 \to b_2}{E \vdash \mathbf{send}\ e_1\ e_2 \to \mathbf{True}} \quad \frac{E \vdash e \to b}{E \vdash \mathbf{receive}\ e \to \mathbf{True}}$$

$$\frac{E \vdash e \to b}{E \vdash \mathbf{select}\ l\ e \to \mathbf{True}} \quad \frac{E \vdash e \to b \quad E(E_i) = b_i}{E \vdash \mathbf{match}\ e\ \mathbf{with}\ \{l_i : e_i\} \to \mathbf{True}} \quad \frac{E \vdash e \to b}{E \vdash \mathbf{fork}\ e \to \mathbf{True}}$$

$$\frac{}{E \vdash C \to \mathbf{False}} \quad \frac{E \vdash e \to b \quad E(C_i) = b_i}{E \vdash \mathbf{case}\ e\ \mathbf{of}\ \{C_i : e_i\} \to b_1}$$

Figura 4.2: Regras para a anotação da árvore sintática

Para cada corpo de função, é necessário percorrer as suas expressões para determinar as que são operações sobre um mónade, ou seja, as que têm alguma expressão de IO. Esta fase é necessária porque durante a fase de anotação das chamadas a funções ainda não existe informação suficiente sobre a função que permita concluir se deve

estar na forma de um mónade.

Assim sendo, geramos código para cada expressão com base na tabela abaixo que é composta pelo valor esperado (anotado na árvore sintática) e pelo valor encontrado pela função de tradução a cada momento da geração de código.

Valor esperado (anotação da árvore sintática)	Valor encontrado (na função de tradução)	Código gerado (Haskell)
False	**False**	e
True	**False**	return e
True	**True**	e
False	**True**	e >>= x → x

Nos casos em que o valor esperado e o valor encontrado são iguais a tradução devolve a expressão inalterada. Quando é esperado que uma expressão e seja uma operação sobre um mónade (valor esperado é **True**) e esta não o é (valor encontrado é **False**) a tradução deve ser return e para colocar a expressão num mónade. No último caso, em que o valor esperado é **False** e o valor encontrado é **True** devemos, ao traduzir, retirar a expressão do mónade: e >>= x → x.

Por exemplo, o código fonte da secção 3.1.5 é traduzido no seguinte código Haskell:

```
1  sendTree !t !c =
2    case t of
3      Leaf → _send "LeafC" c
4      Node x l r →
5        _send "NodeC" c >>=
6        \c1 → _send x c1 >>=
7        \c2 → sendTree l c2 >>=
8        \c3 → sendTree r c3 >>=
9        \c4 → return c4
```

Note-se que, como são enviados e recebidos valores, o código traduzido se encontra num mónade de IO. É ainda importante realçar que após a tradução, os parâmetros (t e c) são precedidos do operador ! que devido à extensão *BangPatterns* torna a função *call-by-value*.

O código que permite receber uma árvore binária de inteiros, presente na secção 3.1.5, é traduzido no seguinte código Haskell:

```
1  receiveTree !c =
2    _receive c >>= \(_x0, _x1) →
3    case _x0 of
4      "LeafC"  → let c1 = _x1 in return (Leaf, c1)
```

```
5         "NodeC"  →
6           let c9 = _x1 in _receive c9 >>=
7           \(x, c2) → receiveTree c2 >>=
8           \(left, c3) → receiveTree c3 >>=
9           \(right, c4) → return (Node x left right, c4)
```

A tradução deste código é similar à do código para enviar uma árvore, visto que, apenas apresenta as operações inversas. No entanto, é importante realçar a tradução da expressão **match** que é traduzida numa operação de **receive** que introduz um par cujos elementos são a etiqueta selecionada e o canal para continuar a comunicação.

O processo de tradução omite a tradução dos tipos das funções do código fonte porque os tipos de sessão não são facilmente traduzidos nos tipos tradicionais do Haskell. Contudo, sabemos que o tipo da função sendTree é:

```
sendTree :: Tree → (MVar a, MVar b) → IO (a, MVar b)
```

e o da função receiveTree é:

```
receiveTree :: (MVar a, MVar b) → IO (Tree, (MVar a,MVar b))
```

Nestes tipos é possível observar que as MVar são independentes e não guardam árvores. Têm como único propósito implementar os canais de comunicação e para o concretizar utilizam a primitiva unsafeCoerce como descrito na secção 4.1.3.

4.2 Validação do compilador

4.2.1 Tecnologias

Foram feitos diversos testes com o objetivo de aferir a robustez do compilador. A infraestrutura que foi implementada permite realizar testes unitários e testes de sistema.

Os testes unitários foram implementados com o intuito de verificar as diversas funções do compilador separadamente e deste modo isolar o comportamento para facilitar a deteção de erros de implementação e garantir que as funções têm as propriedades esperadas. Permitiram verificar diversas propriedades da linguagem, como por exemplo, se os tipos depois de interpretados têm a representação interna correta, se dois tipos são equivalentes (e vice-versa), se o *kinding* de um tipo é o esperado, entre outras propriedades.

Os testes de sistema são testes nos quais o compilador é verificado de um modo mais global, uma vez que, para testar um programa é necessário passar por todas as fases do compilador. Para este tipo de testes é necessário escrever um programa na linguagem FreeST e um ficheiro com a extensão (*.expected*) com o resultado esperado do programa.

Deste modo, quando o programa a testar é compilado e executado, compara-se o resultado esperado com o resultado obtido para verificar se o comportamento é o esperado.

Para implementar esta infraestrutura foram utilizadas diferentes tecnologias: o *HSpec*[1] é uma *framework* para testar programas Haskell que permite interligar o *HUnit*[2] para implementar os testes unitários e ainda o *HPC*[3] para determinar a cobertura dos testes.

4.2.2 Resultados

Os resultados apresentados nesta secção foram obtidos através da ferramenta *HPC* que permite obter informações sobre a cobertura dos testes de um programa Haskell. Esta ferramenta procura verificar até que ponto cada parte do programa foi testada. A verificação pode ser medida em três categorias distintas: as declarações, as alternativas e as expressões.

Declarações *top-level* São relativas à cobertura dos testes para as declarações de funções e de tipos de dados.

Alternativas Está relacionada com a cobertura das alternativas que são as opções que estão disponíveis nas ramificações das expressões **case** e **if** ou ainda das próprias funções quando compostas por casos ou quando têm guardas.

Expressões Diz respeito à cobertura de todas as expressões do código, isto é, ás expressões que são executadas pelos testes efetuados.

A tabela 4.1 apresenta os resultados obtidos através desta ferramenta. Os valores presentes na tabela tornaram possível quantificar o grau de cobertura dos testes. No geral, a cobertura obtida nas definições de *top-level* foi positiva exceto os módulos Types (61%) e TypeEquivalence (82%).

O valor obtido para o módulo Types é mais baixo que os restantes derivado do facto das instâncias da classe **Ord** definidas para os tipos de dados não serem testadas. Os tipos de dados necessitam de ser ordenáveis uma vez que são guardados nos ambientes, isto é, nos mapas do módulo Data.Map.Strict e a função member, disponível neste módulo, requisitar que as chaves dos mapas sejam ordenáveis (member :: **Ord** k ⇒ k → Map k a → **Bool**). No caso do módulo TypeEquivalence o valor é inferior ao restantes porque é difícil implementar testes unitários que verifiquem as funções que traduzem os tipos de sessão em gramáticas independentes do contexto, assim como as árvores de expansão geradas.

A utilização desta ferramenta permitiu guiar os testes de modo a atingir a maior parte do código e ainda descobrir código morto que nunca era executado. Por outro lado, a ferramenta permitiu minimizar a quantidade de erros não detetados. Para um programa com uma cobertura elevada, existe uma maior percentagem de código fonte que é executada

[1] `http://hspec.github.io/`
[2] `https://github.com/hspec/HUnit`
[3] `https://wiki.haskell.org/Haskell_program_coverage`

durante os testes quando comparado com um programa com menor cobertura e, como tal, tem uma menor probabilidade de conter erros não detetados.

Módulos	**Definições Top-Level**		**Alternativas**		**Expressões**	
	%	coberto/total	%	coberto/total	%	coberto/total
CodeGen	100%	12/12	99%	4/4	99%	142/143
DatatypeGen	100%	4/4	100%	2/2	100%	75/75
ExpressionGen	100%	14/14	88%	69/78	82%	884/1071
Compiler	100%	3/3	100%	4/4	83%	46/55
Lexer	100%	22/22	-	0/0	100%	85/85
Parser	100%	48/48	83%	5/6	91%	774/843
TypeParser	100%	22/22	66%	4/6	94%	294/310
PreludeLoader	100%	5/5	-	0/0	100%	76/76
Kinds	94%	16/17	100%	8/8	100%	19/19
Terms	100%	2/2	-	0/0	-	0/0
Types	61%	33/54	97%	45/46	98%	297/301
Kinding	100%	17/17	89%	34/38	79%	223/279
TypeEquivalence	82%	46/56	85%	65/76	89%	645/724
Typing	100%	33/33	82%	72/87	67%	780/1162
TypingState	100%	13/13	-	0/0	95%	85/89
Cobertura Total do programa	90%	290/322	87%	312/355	84%	4425/5232

Tabela 4.1: Cobertura dos testes – *HPC*

Capítulo 5

Conclusão e trabalho futuro

No software concorrente, onde os processos comunicam exclusivamente por troca de mensagens, a comunicação torna-se facilmente complexa devido ao elevado número de mensagens que são trocadas entre os diferentes participantes. Assim sendo, é importante definir abstrações que permitam controlar e estruturar a comunicação intensiva. Os tipos de sessão foram desenvolvidos com este propósito. No entanto, têm limitações na estrutura que impossibilitam definir de forma eficiente estruturas de tipos não lineares.

A linguagem de programação que apresentámos é concorrente e explicitamente tipificada, onde os processos comunicam exclusivamente por troca de mensagens. As mensagens são trocadas em canais síncronos e bidirecionais que são descritos por tipos de sessão independentes do contexto [20]. Graças a estes tipos a linguagem é capaz de transmitir num único canal, por exemplo, tipos de dados recursivos estruturados em forma de árvore com segurança de tipos, entre outros exemplos de tipos não lineares.

Ao longo desta dissertação foi apresentado um exemplo (transmissão de uma árvore binária) que permite descrever os aspetos fulcrais da linguagem e para além disso, concretiza o exemplo apresentado no artigo de Thiemann e Vasconcelos [20] que é determinante para ilustrar a necessidade de utilizar tipos de sessão independentes do contexto.

Como trabalho futuro, temos por objetivo reduzir a verbosidade da linguagem tornando possível abreviar tipos (**type** SendInt = !**Int**). Deste modo, elimina-se a necessidade de os escrever ao longo do código fonte, nomeadamente nos tipos das funções e/ou nas aplicações de tipos. As abreviaturas de tipos diminuem a probabilidade da ocorrência erros derivados da definição de tipos.

Existe ainda o objetivo de fazer inferência de tipos em alguns cenários como por exemplo, as aplicações de tipos e[T] para que eliminar a necessidade de especificar os tipos para as chamadas polimórficas.

Por fim, também temos como objetivo introduzir canais partilhados e o operador de **dualof** para que não seja impreterível definir dois tipos distintos, para tipos duais.

Apêndice A

Código Java - Akka

```
1  public class Main {
2    public static void main(String[] args) {
3      ActorSystem sys = ActorSystem.create("system");
4
5      ActorRef server = sys.actorOf(Props.create(Server.class));
6
7      server.tell(new IntMessage(16, 7), ActorRef.noSender());
8      server.tell(new StringMessage("Hello ", "World"), ActorRef.
           noSender());
9    }
10 }
11
12 public class Server extends AbstractActor{
13   @Override
14   public Receive createReceive() {
15     return new ReceiveBuilder()
16         .match(IntMessage.class, msg → {
17           System.out.println(msg.getFst() + msg.getSnd());
18         })
19         .match(StringMessage.class, msg → {
20           System.out.println(msg.getFst() + msg.getSnd());
21         })
22         .build();
23   }
24 }
25
26 public class StringMessage {
27   private String fst, snd;
28
29   public StringMessage(String fst, String snd) {
30     this.fst = fst;
31     this.snd = snd;
32   }
33
34   public String getFst() {
```

```
35      return fst;
36    }
37
38    public String getSnd() {
39      return snd;
40    }
41  }
42
43  public class IntMessage {
44    private int fst, snd;
45
46    public IntMessage(int fst, int snd) {
47      this.fst = fst;
48      this.snd = snd;
49    }
50
51    public int getFst() {
52      return fst;
53    }
54
55    public int getSnd() {
56      return snd;
57    }
58  }
```

Bibliografia

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3:95–230, 2016.

[3] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[4] Pedro Baltazar, Dimitris Mostrous, and Vasco Thudichum Vasconcelos. Linearly refined session types. In *International Workshop on Linearity*, volume 101 of *EPTCS*, pages 38–49, 2012.

[5] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM*, 34(2):8:1–8:78, 2012.

[6] Søren Christensen, Hans Hüttel, and Colin Stirling. Bisimulation equivalence is decidable for all context-free processes. In *CONCUR '92*, pages 138–147. Springer, 1992.

[7] Juliana Franco and Vasco T. Vasconcelos. A concurrent programming language with refined session types. In *Second International Workshop on Behavioural Types*, pages 33–42, 2013.

[8] Simon Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2):191–225, Nov 2005.

[9] Simon J. Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In *POPL*, pages 299–312. ACM, 2010.

[10] The Go programming language. `http://golang.org/`.

[11] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323 – 364, 1977.

[12] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.

[13] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26:100–106, 1983.

[14] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[15] Petr Jančar and Faron Moller. Techniques for decidability and undecidability of bisimilarity. In *CONCUR'99 Concurrency Theory*, pages 30–45. Springer, 1999.

[16] Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor frameworks for the JVM platform: A comparative analysis. In *PPPJ*, pages 11–20. ACM, 2009.

[17] Matthias Neubauer and Peter Thiemann. An implementation of session types. In *International Symposium on Practical Aspects of Declarative Languages*, pages 56–70. Springer, 2004.

[18] Luca Padovani. Context-free session type inference. In *ESOP*, pages 804–830. Springer, 2017.

[19] Riccardo Pucella and Jesse A Tov. Haskell session types with (almost) no class. In *ACM Sigplan Notices*, volume 44, pages 25–36. ACM, 2008.

[20] Peter Thiemann and Vasco T. Vasconcelos. Context-free session types. In *ICFP*, volume 51, pages 462–475. ACM, 2016.

[21] Vasco T. Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52–70, 2012.

[22] David Walker. Substructural type systems. In *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.