UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

GABRIEL DE SOUZA HAGGSTROM

Projeto e prototipação de uma variação do cálculo-lambda com tipos de sessão

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof. Dr. Rodrigo Machado
Co-orientador: Prof. Dr. Álvaro Freitas Moreira

Porto Alegre
2021

RESUMO

Os tipos de sessão são uma extensão de linguagens de programação que permite verificar, em nível de sistema de tipos, se uma comunicação por troca de mensagens entre processos concorrentes respeita um determinado protocolo. São inspirados na lógica linear e surgiram no contexto do cálculo-π (um cálculo de processos), sendo posteriormente incorporados em variantes do cálculo-λ. Recentemente, tipos de sessão estão sendo integrados em linguagens de programação convencionais como Java, Haskell e outras.
O objetivo deste trabalho é explorar a teoria de tipos de sessão e sua aplicação à programação através do projeto, especificação formal e prototipação de uma variante do cálculo-λ estendida com tipos de sessão e operações para programação concorrente. Através desse cálculo, serão apresentados exemplos de uso e idiomas aplicáveis a linguagens de programação que tenham tais extensões.

Palavras-chave: Linguagens de programação. Semântica formal. Sistemas de tipos. Tipos de sessão. Tipos lineares. Cálculo lambda. Programação concorrente.

Design and prototyping of a variant of lambda-calculus with session types

ABSTRACT

Session types are a programming language extension that allows to check, at type system level, if a message-passing communication between concurrent processes conforms to some given protocol. They take inspiration from linear logic and have emerged in the context of π-calculus (a process calculus), being later embedded in variants of λ-calculus. Recently, session types are being integrated in mainstream programming languages such as Java, Haskell and others.

This work aims to explore the theory of session types and their application to programming through the design, formal specification and prototyping of a variant of λ-calculus extended with session types and operations for concurrent programming. Using this calculus, this work will present examples of use and idioms applicable to programming languages that have such extensions.

Keywords: Programming languages. Formal semantics. Type systems. Session types. Linear types. Lambda calculus. Concurrent programming.

LISTA DE FIGURAS

Figura 3.1 Sintaxe de *kinds*, tipos e termos 31
Figura 3.2 Contextos de variáveis 32
Figura 3.3 Regras de boa-formação de contextos e tipos 33
Figura 3.4 Dualidade de tipos de sessão 33
Figura 3.5 Regras de tipo para a parte puramente funcional da linguagem 35
Figura 3.6 Regras de tipo para operações de concorrência 36
Figura 3.7 Sintaxe de valores, configurações e contextos de avaliação 38
Figura 3.8 Regras de redução de termos 39
Figura 3.9 Regras de equivalência e redução para configurações 40

Figura 4.1 Captura de tela mostrando a interface do interpretador 42
Figura 4.2 Exemplo de programa com a sintaxe do interpretador 43

LISTA DE TABELAS

Tabela 4.1 Relação entre a sintaxe formal e a do interpretador 41

LISTA DE ABREVIATURAS E SIGLAS

ASCII American Standard Code for Information Interchange

HTML HyperText Markup Language

SUMÁRIO

1 INTRODUÇÃO 9
2 EXEMPLOS DE PROGRAMAS 12
2.1 Envio e recebimento 12
2.2 Linearidade 15
2.3 Oferta e seleção de opções 16
2.4 Protocolos recursivos 19
2.5 Conexões cíclicas e deadlocks 20
2.6 Pontos de acesso e condições de corrida 23
2.7 Estado compartilhado e não-determinismo 25
2.8 Estruturas de dados 27
3 ESPECIFICAÇÃO FORMAL DA LINGUAGEM 31
3.1 Sintaxe 31
3.2 Sistema de tipos 31
3.3 Semântica operacional 37
4 INTERPRETADOR 41
4.1 Uso do interpretador 41
4.2 Implementação 44
5 CONCLUSÕES E TRABALHO FUTURO 46
REFERÊNCIAS 48

1 INTRODUÇÃO

Tipos de sessão (*session types*, em inglês) são uma extensão de linguagens de programação que permite estruturar a comunicação entre componentes de sistemas concorrentes e distribuídos. Um tipo de sessão especifica quais mensagens devem ser transmitidas através de um canal de comunicação e em qual ordem, permitindo verificar, através de checagem estática de tipos, se um programa implementa corretamente um determinado protocolo. Por exemplo, o tipo de sessão $?\mathbf{Int}.!\mathbf{Bool}.\mathbf{End}$ especifica um protocolo onde se recebe ($?$) um inteiro, depois se envia ($!$) um booleano e então termina ($\mathbf{End}$).

Os tipos de sessão foram introduzidos por Honda (1993), e posteriormente estendidos por Takeuchi, Honda e Kubo (1994), Honda, Vasconcelos e Kubo (1998) e Yoshida e Vasconcelos (2007). Esses primeiros trabalhos foram desenvolvidos no contexto do cálculo-π (MILNER; PARROW; WALKER, 1992), que é um cálculo de processos. Posteriormente, os tipos de sessão foram incorporados em cálculos funcionais, baseados no cálculo-λ (GAY; VASCONCELOS; RAVARA, 2003; VASCONCELOS; RAVARA; GAY, 2004; VASCONCELOS; GAY; RAVARA, 2006). Os tipos de sessão também têm sido muito estudados no contexto de linguagens orientadas a objetos (DEZANI-CIANCAGLINI et al., 2005; DEZANI-CIANCAGLINI et al., 2006; COPPO; DEZANI-CIANCAGLINI; YOSHIDA, 2007; CAPECCHI et al., 2009).

Inicialmente, os tipos de sessão permitiam especificar apenas protocolos de comunicação entre dois participantes (tipos de sessão binários). Mais tarde, os tipos de sessão foram estendidos para protocolos envolvendo múltiplos participantes (tipos de sessão multiparticipante) (HONDA; YOSHIDA; CARBONE, 2008). Neste trabalho, focaremos apenas em tipos de sessão binários.

Além do desenvolvimento teórico, os tipos de sessão têm sido implementados de forma nativa em linguagens de programação como ATS, Links, MOOL, SePi e SILL, e através de bibliotecas ou ferramentas externas para linguagens de programação populares como C, Java, Scala, Python, Haskell, Erlang, Go, OCaml e Rust (ABCD, 2017). Um exemplo de ferramenta que faz uso de tipos de sessão é Scribble (HONDA et al., 2011; Scribble Team, 2015), uma linguagem para a descrição de protocolos multiparticipante. Ela possui ferramentas de geração de código para Java e Scala, combinando checagem estática e dinâmica para garantir a conformidade ao protocolo.

Os tipos de sessão têm sido estudados em uma grande variedade de aplicações como, por exemplo, protocolos de serviços *web* (W3C, 2005), comunicação entre pro-

cessos em sistemas operacionais (FÄHNDRICH et al., 2006), protocolos de segurança de rede (BONELLI; COMPAGNONI; GUNTER, 2005), e na especificação do comportamento dinâmico de componentes de *software* (VALLECILLO; VASCONCELOS; RAVARA, 2006).

Neste trabalho, será apresentada uma variante do cálculo-λ estendida com tipos de sessão binários e operações primitivas de concorrência e comunicação síncrona entre *threads*. O cálculo será apresentado através da especificação formal e da implementação de um protótipo de interpretador. O objetivo deste cálculo e do interpretador é servirem como ferramentas didáticas para uma apresentação introdutória de tipos de sessão, permitindo explorar a teoria e suas aplicações à programação através de um núcleo de linguagem minimalista.

Como tipos de sessão são um tópico de pesquisa relativamente recente, a disponibilidade de material introdutório sobre o assunto é um pouco escassa (especialmente de material na língua portuguesa). Este trabalho busca tornar o tópico um pouco mais acessível. Para isso, o Capítulo 2 apresenta uma introdução a tipos de sessão, mostrando uma série de exemplos simples escritos na linguagem desenvolvida. Esse capítulo exige do leitor apenas alguma familiaridade com o paradigma de programação funcional. Já o Capítulo 3, que define formalmente a linguagem, requer conhecimentos de especificação formal de linguagens e cálculo-λ simplesmente tipado. Pierce (2002) oferece uma introdução abrangente sobre esses assuntos.

Para suportar tipos de sessão, a linguagem desenvolvida faz uso de um sistema de tipos *linear*. Os sistemas de tipos lineares são baseados na lógica linear, inventada por Girard (1987). Desde sua concepção, a lógica linear era vista como uma possível fundamentação lógica para a concorrência. Várias interpretações da lógica linear como um modelo de computação concorrente foram propostas (ABRAMSKY, 1993; ABRAMSKY, 1994; BELLIN; SCOTT, 1994; ABRAMSKY; GAY; NAGARAJAN, 1996). Honda (1993) criou os tipos de sessão inspirando-se na lógica linear, embora não houvesse inicialmente uma conexão formal entre as duas teorias. Mais tarde, Caires e Pfenning (2010) estabeleceram uma correspondência de Curry-Howard entre a lógica linear intuicionista e tipos de sessão e Wadler (2012) estabeleceu uma correspondência de Curry-Howard entre a lógica linear clássica e tipos de sessão. Este trabalho introduz as noções de linearidade necessárias para a compreensão da linguagem, mas o leitor mais interessado no assunto pode recorrer a Wadler (1993) para uma introdução mais abrangente à lógica linear e ao cálculo-λ linear.

O cálculo desenvolvido neste trabalho foi influenciado em parte pelo cálculo GV, introduzido por Wadler (2012) e estendido por Lindley e Morris (2015). O cálculo GV possui uma conexão formal precisa com a lógica linear, o que lhe confere garantias fortes como terminação, determinismo e ausência de *deadlocks*. Essas garantias, porém, vêm ao preço de termos um modelo de concorrência bastante restritivo, no qual não é possível modelar, por exemplo, protocolos repetitivos, condições de corrida e conexões cíclicas entre processos. O cálculo deste trabalho adiciona novos mecanismos de conexão entre processos, como *pontos de acesso* (*access points*), baseando-se em trabalhos como os de Gay e Vasconcelos (2010) e Lindley e Morris (2017), o que permite expressar conexões cíclicas e condições de corrida. O cálculo também adiciona recursão geral através de uma construção "let rec". Esses mecanismos fornecem maior flexibilidade na construção de programas concorrentes, ao custo de introduzir a possibilidade de não-terminação, não-determinismo e *deadlocks*.

O trabalho é estruturado da seguinte forma: o Capítulo 2 fornece uma visão geral da linguagem através de uma série de exemplos, servindo como uma introdução a tipos de sessão. O Capítulo 3 define formalmente a sintaxe, sistema de tipos e semântica operacional da linguagem. O Capítulo 4 descreve a implementação do interpretador da linguagem, e como ele pode ser usado. Por fim, o Capítulo 5 apresenta conclusões e sugestões de trabalho futuro.

2 EXEMPLOS DE PROGRAMAS

Neste capítulo, será apresentada uma série de exemplos mostrando os principais aspectos da linguagem desenvolvida. Nos primeiros exemplos, assumimos que temos definidos os tipos Nat e $Bool$ e operações sobre esses tipos. Na Seção 2.8, será mostrado como esses tipos podem ser definidos usando as construções primitivas da linguagem.

Os arquivos com o código de cada exemplo estão disponíveis *on-line*. Cada arquivo inclui as definições de Nat e $Bool$ necessárias para rodar o exemplo. Veja a Seção 4.1, que explica o uso do interpretador.

2.1 Envio e recebimento

Começamos com um protocolo simples entre cliente e servidor, onde o cliente envia um número natural para o servidor, então o servidor responde com um valor booleano e, por fim, o protocolo encerra. O tipo de sessão que define o protocolo do ponto de vista do cliente seria:

$$Cliente = !Nat.?Bool.\mathbf{End}$$

Um tipo de sessão $!A.B$ define um protocolo no qual enviamos um valor do tipo A e então continuamos como B. Dualmente, um tipo de sessão $?A.B$ define um protocolo no qual recebemos um valor do tipo A e então continuamos como B. O tipo de sessão $\mathbf{End}$ representa um protocolo que se encerra sem nenhuma ação.

O tipo de sessão do servidor então é definido como:

$$Servidor = ?Nat.!Bool.\mathbf{End}$$

Os tipos $Servidor$ e $Cliente$ são *duais* um do outro: quando o cliente envia um Nat, o servidor recebe um Nat; quando o servidor envia um $Bool$, o cliente recebe um $Bool$; e quando o cliente encerra, o servidor encerra ao mesmo tempo. O dual de um tipo de sessão A é denotado por $\overline{A}$. Portanto, as seguintes igualdades são válidas:

$$Servidor = \overline{Cliente}$$
$$Cliente = \overline{Servidor}$$

A noção de dualidade é essencial em tipos de sessão, pois garante que ambos os lados da

comunicação interagem entre si de forma compatível, obedecendo ao mesmo protocolo. Em tipos de sessão multiparticipante, a noção correspondente é chamada de *coerência* (CARBONE et al., 2016). A operação de dualidade será definida de forma precisa na Seção 3.2 (Figura 3.4).

Cliente e servidor se comunicam através de um *canal*. Um canal pode ser imaginado como um "fio"que conecta os dois participantes da comunicação. Um canal possui duas *extremidades* (*endpoints*, em inglês), que são as duas "pontas"do fio. Com tipos de sessão multiparticipante, um canal pode conectar diversos participantes, tendo uma extremidade para cada participante.

Uma implementação de cliente pode ser definida como uma função que recebe uma das extremidades do canal, do tipo $Cliente$, que será usada para se comunicar com o servidor:

$$
\begin{array}{l}
\textbf{let un } cliente = \\
\quad \lambda \textbf{lin } c_0 : Cliente. \\
\qquad \textbf{let lin } c_1 = \textbf{send } 0\ c_0 \textbf{ in} \\
\qquad \textbf{let } \{b, c_2\} = \textbf{receive } c_1 \textbf{ in} \\
\qquad \textbf{let } \{\} = \textbf{close } c_2 \textbf{ in} \\
\qquad b \\
\textbf{in } \ldots
\end{array}
$$

As palavras reservadas **lin** e **un** declaram variáveis como sendo *lineares* ou *irrestritas*, respectivamente. Veremos o que isso significa na Seção 2.2. Por enquanto, essas anotações podem ser ignoradas.

O comando **send** envia um valor do tipo A através de uma extremidade de canal do tipo de sessão $!A.B$, e retorna uma nova extremidade de canal, do tipo de sessão B, que é usada para prosseguir com a comunicação. Como a extremidade c_0 é do tipo $Cliente = !Nat.?Bool.\mathbf{End}$, o valor a ser enviado deve ser do tipo Nat (no caso, 0) e a extremidade retornada é do tipo $?Bool.\mathbf{End}$, a qual é atribuída à variável c_1.

O comando **receive** realiza, através de uma extremidade de canal do tipo de sessão $?A.B$, o recebimento de um valor do tipo A. A operação retorna um valor do tipo $A \times B$, ou seja, um par, onde o primeiro componente é o valor recebido e o segundo componente é uma nova extremidade de canal, do tipo de sessão B, que é usada para prosseguir com a comunicação. Como c_1 é do tipo $?Bool.\mathbf{End}$, o valor resultante da operação é um par do tipo $Bool \times \mathbf{End}$, cujos componentes são atribuídos às variáveis b e c_2 através da sintaxe de desconstrução de pares "$\textbf{let } \{b, c_2\} = \ldots \textbf{ in } \ldots$".

O comando **close** encerra a comunicação através de uma extremidade de canal do

tipo $\mathbf{End}$ (que é o tipo de c_2). A operação retorna o valor $\{\}$ (uma tupla vazia), do tipo $\mathbf{1}$, o qual pode ser consumido com o desconstrutor "$\mathbf{let}\ \{\} = \ldots\ \mathbf{in}\ \ldots$". Por fim, o cliente retorna o booleano b que havia sido recebido através do $\mathbf{receive}$.

Usando esses mesmos comandos, podemos definir um servidor como uma função que recebe a outra extremidade do canal, do tipo $Servidor$:

$$
\begin{array}{l}
\mathbf{let\ un}\ servidor = \\
\quad \lambda \mathbf{lin}\ s_0 : Servidor. \\
\qquad \mathbf{let}\ \{n, s_1\} = \mathbf{receive}\ s_0\ \mathbf{in} \\
\qquad \mathbf{let\ lin}\ s_2 = \mathbf{send}\ (n == 0)\ s_1\ \mathbf{in} \\
\qquad \mathbf{close}\ s_2 \\
\mathbf{in}\ \ldots
\end{array}
$$

Em resumo, o servidor recebe um natural n, testa se n é igual a 0, envia o booleano resultante do teste, e encerra, retornando o valor $\{\}$, do tipo $\mathbf{1}$, resultante do $\mathbf{close}$.

Para executar o cliente e o servidor em paralelo, conectados através de um canal, usamos a operação $\mathbf{fork}$:

$$
\begin{array}{l}
\mathbf{let\ lin}\ c = \mathbf{fork}\ servidor\ \mathbf{in} \\
cliente\ c
\end{array}
$$

O comando $\mathbf{fork}$ toma como parâmetro uma função linear (veremos sobre linearidade na Seção 2.2), a qual recebe uma extremidade de canal de um tipo de sessão A e devolve $\{\}$, do tipo $\mathbf{1}$. O que o comando faz é disparar a função como uma nova *thread*, em paralelo, e criar um canal de comunicação entre a *thread* mãe (a que executa o $\mathbf{fork}$) e a *thread* filha, passando uma das extremidades, do tipo A, para a função, e retornando a outra extremidade, do tipo $\overline{A}$, para a *thread* mãe, como resultado da operação. A função passada para o $\mathbf{fork}$ deve retornar um valor descartável $\{\}$ porque apenas a *thread* principal (aquela que inicia a execução do programa) retorna o resultado final do programa, enquanto os valores retornados pelas *threads* filhas são descartados. O tipo A deve ser um tipo de sessão, pois tipos de dados como Nat e $Bool$ não possuem dual.

Como $servidor$ é uma função de $Servidor$ para $\mathbf{1}$, a extremidade passada para essa função é do tipo $Servidor$, e a extremidade resultante do $\mathbf{fork}$ é do tipo $Cliente$, a qual é atribuída à variável c, que é passada como argumento para a função $cliente$. A partir daí, cliente e servidor estarão rodando em paralelo e se comunicando através do canal criado pelo $\mathbf{fork}$. O cliente envia o natural 0 para o servidor, o servidor testa se

o valor é igual a 0 e envia o resultado $true$ para o cliente, ambos encerram o canal de comunicação e, por fim, o cliente retorna o booleano $true$ recebido, como resultado final do programa.

2.2 Linearidade

Para suportar tipos de sessão, a linguagem faz uso de um sistema de tipos *linear*. Para entendermos o que isso significa, e por que isso é necessário, vejamos o seguinte exemplo de cliente:

$$
\begin{array}{l}
\textbf{let un } clienteErrado = \\
\quad \lambda \textbf{lin } c_0 : Cliente. \\
\qquad \textbf{let lin } c_1 = \textbf{send } 0\ c_0 \textbf{ in} \\
\qquad \textbf{let lin } c_2 = \textbf{send } 1\ c_0 \textbf{ in} \\
\qquad false \\
\textbf{in } \ldots
\end{array}
$$

Veja que os tipos dos argumentos passados para as duas operações de **send** estão corretos. Mas o protocolo claramente não está sendo seguido: o cliente envia dois naturais, quando deveria enviar apenas um, e ele nunca recebe o booleano do servidor nem encerra o canal.

Isso acontece porque a variável c_0, cujo tipo especifica que a próxima operação deve ser o envio de um Nat, está sendo usada duas vezes, e as variáveis c_1 e c_2, do tipo $?Bool.\mathbf{End}$, que especificam o recebimento de um $Bool$ e encerramento do canal, nunca são usadas. Para resolver esse problema, devemos garantir que cada uma dessas variáveis seja usada *exatamente uma vez*, para que cada passo do protocolo seja executado *exatamente uma vez*. Isso é feito declarando essas variáveis como lineares.

Observe que as variáveis do exemplo acima já foram declaradas como lineares, usando a palavra reservada **lin**. O sistema de tipos garante que uma variável linear é usada exatamente uma vez. Se a variável linear não for usada, ou se for usada mais de uma vez, o programa é considerado mal-tipado. Portanto, o exemplo acima *é mal-tipado*.

Vejamos novamente o exemplo correto de cliente:

$$
\begin{array}{l}
\textbf{let un } cliente = \\
\quad \lambda \textbf{lin } c_0 : Cliente. \\
\qquad \textbf{let lin } c_1 = \textbf{send } 0\ c_0 \textbf{ in} \\
\qquad \textbf{let } \{b, c_2\} = \textbf{receive } c_1 \textbf{ in} \\
\qquad \textbf{let } \{\} = \textbf{close } c_2 \textbf{ in} \\
\qquad b \\
\textbf{in } \ldots
\end{array}
$$

As variáveis c_0 e c_1 foram declaradas como lineares. Em uma desconstrução de par "$\textbf{let } \{x, y\} = ... \textbf{ in } ...$", as variáveis x e y são sempre lineares, portanto as variáveis b e c_2 do exemplo também são lineares. Observe que cada uma dessas variáveis é usada exatamente uma vez e, de fato, o código acima é bem-tipado, do tipo $Cliente \multimap Bool$, isto é, uma função linear de $Cliente$ para $Bool$. O sistema de tipos será definido de forma mais precisa na Seção 3.2.

A variável $cliente$ é declarada como irrestrita, usando a palavra reservada **un** (de *unrestricted*). Isso significa que a variável não é linear e poderá ser usada zero ou mais vezes no restante do programa.

O leitor pode recorrer a Wadler (1993) para uma introdução mais abrangente a sistemas de tipos lineares.

2.3 Oferta e seleção de opções

Além de envio e recebimento, os tipos de sessão também permitem especificar protocolos onde um dos participantes realiza uma escolha dentre um conjunto de opções ofertadas. Por simplicidade, a linguagem considera apenas o caso binário, onde há duas opções.

Para exemplificar escolhas de opções, vamos estender o exemplo anterior, considerando um protocolo onde o servidor oferece para o cliente duas opções: testar se um natural é igual a zero, como no exemplo anterior; ou realizar uma adição, onde o cliente envia dois naturais para o servidor, e o servidor responde com a soma destes. O tipo de sessão do cliente seria:

$$Cliente = \oplus\{!Nat.?Bool.\mathbf{End},\ !Nat.!Nat.?Nat.\mathbf{End}\}$$

Um tipo de sessão $\oplus\{A,\ B\}$ define um protocolo onde se deve selecionar uma das opções: **left** ou **right**. Ao selecionar **left**, o protocolo continua de acordo com o tipo de sessão A. Ao selecionar **right**, o protocolo segue de acordo com o tipo de sessão B. No exemplo, a opção **left** será o teste de zero e a opção **right** será a adição. O tipo do servidor é o dual de $Cliente$:

$$Servidor = \&\{?Nat.!Bool.\mathbf{End},\ ?Nat.?Nat.!Nat.\mathbf{End}\}$$

Um tipo de sessão $\&\{A,\ B\}$ define um protocolo onde se ofertam duas opções para o participante na outra extremidade do canal: **left** e **right**. Caso a opção selecionada pelo outro participante seja **left**, o protocolo continua de acordo com o tipo de sessão A. Se a escolha for **right**, o protocolo segue de acordo com o tipo de sessão B.

Uma implementação de cliente pode ser:

$$\begin{array}{l}
\mathbf{let\ un}\ cliente = \\
\quad \lambda\mathbf{lin}\ c_0 : Cliente. \\
\qquad \mathbf{let\ lin}\ c_1 = \mathbf{select\ right}\ c_0\ \mathbf{in} \\
\qquad \mathbf{let\ lin}\ c_2 = \mathbf{send}\ 1\ c_1\ \mathbf{in} \\
\qquad \mathbf{let\ lin}\ c_3 = \mathbf{send}\ 1\ c_2\ \mathbf{in} \\
\qquad \mathbf{let}\ \{n, c_4\} = \mathbf{receive}\ c_3\ \mathbf{in} \\
\qquad \mathbf{let}\ \{\} = \mathbf{close}\ c_4\ \mathbf{in} \\
\qquad n \\
\mathbf{in}\ \ldots
\end{array}$$

O comando **select** seleciona uma opção através de uma extremidade de canal de um tipo $\oplus\{A,\ B\}$, e retorna uma nova extremidade, do tipo A se a opção selecionada for **left**, ou B se a escolha for **right**. No exemplo, como a extremidade c_0 é do tipo $\oplus\{!Nat.?Bool.\mathbf{End},\ !Nat.!Nat.?Nat.\mathbf{End}\}$ e a opção selecionada é **right**, o tipo de c_1 é o do protocolo de adição, $!Nat.!Nat.?Nat.\mathbf{End}$. O restante do protocolo é implementado usando os mesmos comandos vistos no exemplo anterior.

A implementação do servidor pode ser definida como:

$$
\begin{array}{l}
\textbf{let un } servidor = \\
\quad \lambda \textbf{lin } s_0 : Servidor. \\
\qquad \textbf{branch } s_0 \textbf{ of} \\
\qquad\quad \textbf{left } s_1 \mapsto \\
\qquad\qquad \textbf{let } \{n, s_2\} = \textbf{receive } s_1 \textbf{ in} \\
\qquad\qquad \textbf{let lin } s_3 = \textbf{send } (n == 0)\ s_2 \textbf{ in} \\
\qquad\qquad \textbf{close } s_3\ ; \\
\\
\qquad\quad \textbf{right } s_1 \mapsto \\
\qquad\qquad \textbf{let } \{n_1, s_2\} = \textbf{receive } s_1 \textbf{ in} \\
\qquad\qquad \textbf{let } \{n_2, s_3\} = \textbf{receive } s_2 \textbf{ in} \\
\qquad\qquad \textbf{let lin } s_4 = \textbf{send } (n_1 + n_2)\ s_3 \textbf{ in} \\
\qquad\qquad \textbf{close } s_4 \\
\textbf{in } \ldots
\end{array}
$$

Uma expressão "$\textbf{branch } \ldots \textbf{ of left } x \mapsto \ldots;\ \textbf{right } y \mapsto \ldots$" oferta as opções **left** e **right** através de uma extremidade de canal de um tipo $\&\{A,\ B\}$. Como a escolha é feita externamente, a implementação do protocolo precisa considerar as duas possibilidades: um dos ramos do **branch** é executado se a opção escolhida for **left**, ou o outro se a escolha for **right**. No ramo da esquerda, uma variável linear x é introduzida, do tipo de sessão A, a qual é usada para implementar o protocolo definido por A. De forma semelhante, uma variável linear y é introduzida no ramo da direita, do tipo de sessão B. Ambos os ramos devem retornar algo do mesmo tipo, senão não é possível inferir o tipo da expressão **branch** como um todo. No exemplo, a extremidade s_0 é do tipo $\&\{?Nat.!Bool.\mathbf{End},\ ?Nat.?Nat.!Nat.\mathbf{End}\}$, portanto a variável s_1 do primeiro ramo é do tipo $?Nat.!Bool.\mathbf{End}$, e a variável de mesmo nome no segundo ramo é do tipo $?Nat.?Nat.!Nat.\mathbf{End}$.

Cliente e servidor são conectados através do comando **fork**, exatamente como no primeiro exemplo. O cliente seleciona a opção **right** (adição), fazendo com que o servidor execute o segundo ramo do **branch**. O cliente envia os naturais 1 e 1 e recebe a soma 2, a qual é retornada como resultado do programa.

2.4 Protocolos recursivos

Vamos agora adicionar repetição ao protocolo. O cliente terá duas opções: o teste de zero (**left**), onde o cliente envia um natural, recebe um booleano e então o protocolo repete, oferecendo novamente as duas opções; ou a opção de encerrar (**right**), que é simplesmente o protocolo **End**. Assim, o cliente pode selecionar a opção **left** repetidas vezes, e por fim selecionar **right** para encerrar a comunicação. A seleção de opções é feita como no exemplo anterior. Para expressar a repetição, usamos tipos recursivos. Os tipos do cliente e do servidor (duais, como sempre) são os seguintes:

$$Cliente = \mathbf{rec}\ X : *_S.\ \oplus \{!Nat.?Bool.X,\ \mathbf{End}\}$$
$$Servidor = \mathbf{rec}\ X : *_S.\ \& \{?Nat.!Bool.X,\ \mathbf{End}\}$$

Um tipo de sessão $\mathbf{rec}\ X : *_S.A$ define um protocolo recursivo, onde o protocolo definido pelo tipo de sessão A pode fazer referência a si mesmo através da variável de tipo X. Pierce (2002, Capítulo 20) apresenta uma introdução abrangente sobre tipos recursivos. A anotação $*_S$ declara o tipo recursivo referenciado por X como um tipo de sessão. Para tipos recursivos que não são de sessão, usa-se $*_{NS}$ (de *Non-Session*).

Um exemplo de implementação de cliente seria:

$$\begin{array}{l}
\mathbf{let\ un}\ cliente = \\
\quad \lambda\mathbf{lin}\ c_0 : Cliente. \\
\qquad \mathbf{let\ lin}\ c_1 = \mathbf{select\ left}\ (\mathbf{unfold}\ c_0)\ \mathbf{in} \\
\qquad \mathbf{let\ lin}\ c_2 = \mathbf{send}\ 0\ c_1\ \mathbf{in} \\
\qquad \mathbf{let}\ \{b_0, c_3\} = \mathbf{receive}\ c_2\ \mathbf{in} \\
\qquad \mathbf{let\ lin}\ c_4 = \mathbf{select\ left}\ (\mathbf{unfold}\ c_3)\ \mathbf{in} \\
\qquad \mathbf{let\ lin}\ c_5 = \mathbf{send}\ 1\ c_4\ \mathbf{in} \\
\qquad \mathbf{let}\ \{b_1, c_6\} = \mathbf{receive}\ c_5\ \mathbf{in} \\
\qquad \mathbf{let\ lin}\ c_7 = \mathbf{select\ right}\ (\mathbf{unfold}\ c_6)\ \mathbf{in} \\
\qquad \mathbf{let}\ \{\} = \mathbf{close}\ c_7\ \mathbf{in} \\
\qquad b_0\ or\ b_1 \\
\mathbf{in}\ \ldots
\end{array}$$

O cliente começa selecionando a opção **left** (teste de zero). O comando **select** espera algo de um tipo $\oplus\{A,\ B\}$, mas o tipo de c_0 é um tipo recursivo. Por isso, a operação **unfold** é usada. Ela recebe algo de um tipo recursivo $\mathbf{rec}\ X.A$ (anotação $*_S$ ou $*_{NS}$ omitida) e devolve o *unfolding*, ou seja, algo do tipo $A[X := \mathbf{rec}\ X.A]$, que é a substituição de

X pelo próprio tipo recursivo em A. Como c_0 é do tipo $\mathbf{rec}\ X.\oplus\{!Nat.?Bool.X,\ \mathbf{End}\}$, o tipo de $\mathbf{unfold}\ c_0$ é $\oplus\{!Nat.?Bool.(\mathbf{rec}\ X.\oplus\{!Nat.?Bool.X,\ \mathbf{End}\}),\ \mathbf{End}\}$, exatamente da forma esperada pelo $\mathbf{select}$. O mesmo ocorre nos outros dois usos de $\mathbf{unfold}$.

O servidor pode ser implementado da seguinte forma:

$$
\begin{array}{l}
\mathbf{let\ rec}\ servidor : Servidor \multimap \mathbf{1} = \\
\quad \lambda \mathbf{lin}\ s_0 : Servidor. \\
\qquad \mathbf{branch\ unfold}\ s_0\ \mathbf{of} \\
\qquad\quad \mathbf{left}\ s_1 \mapsto \\
\qquad\qquad \mathbf{let}\ \{n, s_2\} = \mathbf{receive}\ s_1\ \mathbf{in} \\
\qquad\qquad \mathbf{let\ lin}\ s_3 = \mathbf{send}\ (n == 0)\ s_2\ \mathbf{in} \\
\qquad\qquad servidor\ s_3\ ; \\
\\
\qquad\quad \mathbf{right}\ s_1 \mapsto \\
\qquad\qquad \mathbf{close}\ s_1 \\
\mathbf{in}\ \ldots
\end{array}
$$

Observe que, para implementar o protocolo recursivo, a definição acima faz também uso de recursão, usando uma expressão $\mathbf{let\ rec}$. O tipo da variável recursiva precisa ser declarado explicitamente para que possa ser inferido no corpo da definição. A variável definida é irrestrita, tanto no corpo da definição quanto no restante do programa.

De forma semelhante ao cliente, o servidor aplica $\mathbf{unfold}$ para poder usar a extremidade de canal na expressão $\mathbf{branch}$. E como nos exemplos anteriores, cliente e servidor são conectados através de um $\mathbf{fork}$.

2.5 Conexões cíclicas e deadlocks

Nos exemplos anteriores, cliente e servidor eram conectados usando $\mathbf{fork}$, que é a forma mais simples de conectar duas *threads* através de um canal. O comando $\mathbf{fork}$ executa duas ações: dispara uma nova *thread* em paralelo com a atual; e cria um novo canal que conecta as *threads* mãe e filha.

Com o $\mathbf{fork}$, porém, não é possível formar estruturas cíclicas de conexão entre *threads*. Uma vantagem disso é que, sem ciclos, não há a ocorrência de *deadlocks*. Wadler (2012) fez uso dessa abordagem para garantir ausência de *deadlocks* em seus cálculos. A desvantagem, obviamente, é que essa restrição exclui qualquer aplicação prática que faça uso de conexões cíclicas.

Como a linguagem definida neste trabalho visa suportar a programação concorrente de uma forma mais geral, incluindo a possibilidade de conexões cíclicas, ela contorna essa restrição do comando **fork** introduzindo mais dois comandos: **new session** e **spawn**. Esses dois comandos permitem realizar as duas ações do **fork** de forma separada: o **new session** apenas cria um canal e o **spawn** apenas dispara uma nova *thread* em paralelo. Podemos usar esses dois comandos para conectar cliente e servidor como nos exemplos anteriores:

$$\begin{array}{l}\textbf{let } \{s, c\} = \textbf{new session } [Servidor] \textbf{ in} \\ \textbf{let } \{\} = \textbf{spawn } (servidor\ s) \textbf{ in} \\ cliente\ c\end{array}$$

O comando **new session** recebe um tipo de sessão A como parâmetro e cria um canal, retornando as duas extremidades do canal na forma de um par do tipo $A \times \overline{A}$. Portanto, no exemplo, o tipo de s é $Servidor$ e o tipo de c é $Cliente$.

O comando **spawn** recebe como parâmetro uma expressão do tipo $\mathbf{1}$ e dispara uma nova *thread* que avalia essa expressão em paralelo. No exemplo, a nova *thread* rodará a expressão $servidor\ s$. Como o **spawn** não cria um canal, as extremidades s e c do canal criado com o **new session** são passadas manualmente para as funções $servidor$ e $cliente$.

Para vermos como uma conexão cíclica pode ser criada, vamos considerar uma simplificação do exemplo criado por Milner (1989) e também apresentado por Dardha e Gay (2018), de um *escalonador cíclico*. Um conjunto de processos faz uso de um recurso que somente pode ser usado por um processo de cada vez, como, por exemplo, o acesso a um meio de transmissão em rede. Para controlar o acesso ao recurso, os processos se conectam entre si em uma topologia de anel e usam um *token*, que é passado de um processo para o outro de forma circular. Quando um processo recebe o *token*, ele utiliza o recurso por uma quantidade finita de tempo, e então repassa o *token* para o próximo processo. O protocolo de transmissão do *token* pode ser definido da seguinte forma (onde $Token$ pode ser simplesmente o tipo $\mathbf{1}$):

$$RecebeToken = \textbf{rec } X : *_S.\ ?Token.X$$
$$EnviaToken = \textbf{rec } X : *_S.\ !Token.X$$

Cada *thread* recebe como parâmetro duas extremidades de canal, para se comuni-

car com as duas *threads* vizinhas. A *thread* recebe o *token* de um vizinho através de uma das extremidades, executa sua tarefa por uma quantia finita de tempo, envia o *token* para o outro vizinho através da outra extremidade, e repete o processo:

$$
\begin{array}{l}
\textbf{let rec}\ recebeEnvia : RecebeToken \multimap EnviaToken \multimap \mathbf{1} = \\
\quad \lambda\textbf{lin}\ r_0 : RecebeToken. \\
\quad \lambda\textbf{lin}\ e_0 : EnviaToken. \\
\qquad \textbf{let}\ \{t, r_1\} = \textbf{receive}\ (\textbf{unfold}\ r_0)\ \textbf{in} \\
\qquad \text{\{- Executa tarefa -\}} \\
\qquad \textbf{let lin}\ e_1 = \textbf{send}\ t\ (\textbf{unfold}\ e_0)\ \textbf{in} \\
\qquad recebeEnvia\ r_1\ e_1 \\
\textbf{in}\ \ldots
\end{array}
$$

Porém, é necessário que uma das *threads* comece enviando o *token*:

$$
\begin{array}{l}
\textbf{let rec}\ enviaRecebe : Token \multimap RecebeToken \multimap EnviaToken \multimap \mathbf{1} = \\
\quad \lambda\textbf{lin}\ t_0 : Token. \\
\quad \lambda\textbf{lin}\ r_0 : RecebeToken. \\
\quad \lambda\textbf{lin}\ e_0 : EnviaToken. \\
\qquad \text{\{- Executa tarefa -\}} \\
\qquad \textbf{let lin}\ e_1 = \textbf{send}\ t_0\ (\textbf{unfold}\ e_0)\ \textbf{in} \\
\qquad \textbf{let}\ \{t_1, r_1\} = \textbf{receive}\ (\textbf{unfold}\ r_0)\ \textbf{in} \\
\qquad enviaRecebe\ t_1\ r_1\ e_1 \\
\textbf{in}\ \ldots
\end{array}
$$

Agora, conectamos quatro *threads* em uma topologia de anel:

$$
\begin{array}{l}
\textbf{let}\ \{r_0, e_0\} = \textbf{new session}\ [RecebeToken]\ \textbf{in} \\
\textbf{let lin}\ r_1 = \textbf{fork}\ (recebeEnvia\ r_0)\ \textbf{in} \\
\textbf{let lin}\ r_2 = \textbf{fork}\ (recebeEnvia\ r_1)\ \textbf{in} \\
\textbf{let lin}\ r_3 = \textbf{fork}\ (recebeEnvia\ r_2)\ \textbf{in} \\
enviaRecebe\ \{\}\ r_3\ e_0
\end{array}
$$

Cada operação de **fork** cria um canal e passa a extremidade do tipo $EnviaToken$ para a nova *thread*. Como cada *thread* precisa de duas extremidades, a extremidade do tipo $RecebeToken$ é passada manualmente. A extremidade retornada pelo primeiro **fork** é passada para a segunda *thread*, a retornada pelo segundo **fork** é passada para a terceira *thread* e assim sucessivamente. Para conectar a última *thread* com a primeira, porém, é necessário fazer uso do **new session**.

Com todas as *threads* conectadas, a *thread* $enviaRecebe$ começa enviando o *token*, e a partir daí o *token* circula infinitamente pelo anel. O programa nunca termina.

Os tipos de sessão garantem que um *deadlock* não ocorre *dentro de uma única sessão*: como ambos os lados do canal se comunicam de forma dual, não é possível que ocorram situações como os dois participantes tentando enviar ou receber ao mesmo tempo, ou um dos participantes tentando se comunicar enquanto o outro já encerrou a comunicação. Porém, é possível que ocorram *deadlocks* quando os participantes se conectam de forma cíclica e intercalam *duas ou mais sessões*. No exemplo acima, cada *thread* se comunica através de dois canais, mas não ocorre *deadlock*. Porém, se trocarmos a *thread* $enviaRecebe$ por outra *thread* $recebeEnvia$

$$
\begin{array}{l}
\textbf{let } \{r_0, e_0\} = \textbf{new session } [RecebeToken] \textbf{ in} \\
\textbf{let lin } r_1 = \textbf{fork } (recebeEnvia\ r_0) \textbf{ in} \\
\textbf{let lin } r_2 = \textbf{fork } (recebeEnvia\ r_1) \textbf{ in} \\
\textbf{let lin } r_3 = \textbf{fork } (recebeEnvia\ r_2) \textbf{ in} \\
recebeEnvia\ r_3\ e_0
\end{array}
$$

o programa entra em *deadlock*, pois todas as *threads* iniciam esperando receber um *token*, mas nenhuma delas o envia.

2.6 Pontos de acesso e condições de corrida

Além de **fork** e **new session**, a linguagem apresenta uma terceira forma, ainda mais flexível, de criar canais: *pontos de acesso*. Podemos usar pontos de acesso para conectar cliente e servidor como nos primeiros exemplos:

$$
\begin{array}{l}
\textbf{new access } [Servidor]\ acessoSrv, acessoClt \textbf{ in} \\
\textbf{let } \{\} = \textbf{spawn } (servidor\ (\textbf{accept } acessoSrv)) \textbf{ in} \\
cliente\ (\textbf{request } acessoClt)
\end{array}
$$

O comando **new access** cria um novo ponto de acesso. Ele recebe como parâmetro um tipo de sessão A e introduz duas variáveis: uma do tipo $\mathbf{Accept}\ A$, que pode ser usada pelo comando **accept** para obter uma extremidade de canal do tipo A; e outra do tipo $\mathbf{Request}\ \overline{A}$, que pode ser usada pelo comando **request** para obter uma extremidade de canal do tipo $\overline{A}$. Ambas as variáveis fazem referência ao mesmo ponto de acesso. Essas variáveis são irrestritas, ou seja, podem ser usadas zero ou mais vezes. Quando temos

uma *thread* realizando **accept** e outra realizando **request** a partir de um mesmo ponto de acesso, um canal de sessão entre elas é criado.

No exemplo, a variável $acessoSrv$ é do tipo $\mathbf{Accept}\ Servidor$ e a variável $acessoClt$ é do tipo $\mathbf{Request}\ Cliente$. Quando a *thread* criada pelo **spawn** executa $\mathbf{accept}\ acessoSrv$ e a *thread* principal executa $\mathbf{request}\ acessoClt$, um canal é criado, a extremidade do tipo $Servidor$ é retornada pelo **accept** e a extremidade do tipo $Cliente$ é retornada pelo **request**. A partir daí, cliente e servidor estão conectados como nos exemplos anteriores. Um ponto de acesso também poderia ser usado para conectar as *threads* do exemplo do escalonador cíclico, substituindo o **new session**.

A decisão de quem realiza **accept** e quem realiza **request** é arbitrária: como as duas operações são simétricas, poderíamos igualmente ter o servidor realizando **request** e o cliente realizando **accept**, bastando para isso trocar o parâmetro $Servidor$ no comando **new access** pelo seu dual, $Cliente$.

O que diferencia os pontos de acesso das operações de criação de canal vistas anteriormente é que eles introduzem *condições de corrida*. Segundo Praun (2011), uma condição de corrida ocorre na execução de um programa paralelo quando duas ou mais *threads* acessam um recurso em comum, e a ordem dos acessos depende do *timing*, isto é, do progresso de cada *thread*. Nesse caso, o recurso em comum é o ponto de acesso. Quando duas ou mais *threads* executam o mesmo tipo de operação (**accept** ou **request**) a partir de um mesmo ponto de acesso, elas estão competindo pelo acesso: a primeira *thread* a executar **accept** se conecta com a primeira *thread* a executar **request**.

Para vermos isso, consideremos o exemplo de um servidor *multithread*, onde são conectadas várias *threads* de servidor e cliente, do primeiro exemplo (Seção 2.1), a partir de um único ponto de acesso. Os clientes usarão o ponto de acesso para se conectar ao servidor. O protocolo continua o mesmo. A função $servidor$ também é a mesma (teste de zero), mas teremos uma *thread* que fica em um *loop* infinito realizando **accept**s e disparando uma nova *thread* de servidor para atender cada cliente que realiza um **request**:

$$
\begin{array}{l}
\mathbf{let\ rec}\ loopServidor : (\mathbf{Accept}\ Servidor) \to \mathbf{1} = \\
\quad \lambda \mathbf{un}\ acessoSrv : \mathbf{Accept}\ Servidor. \\
\qquad \mathbf{let\ lin}\ s = \mathbf{accept}\ acessoSrv\ \mathbf{in} \\
\qquad \mathbf{let}\ \{\} = \mathbf{spawn}\ (servidor\ s)\ \mathbf{in} \\
\qquad loopServidor\ acessoSrv \\
\mathbf{in}\ \ldots
\end{array}
$$

Veja que o tipo declarado para a função usa a seta $\rightarrow$ em vez de $\multimap$, indicando que a função é irrestrita, ou seja, que foi introduzida com $\lambda\mathbf{un}$.

Definimos então dois clientes:

$$
\begin{array}{ll}
\textbf{let un } cliente_1 = & \textbf{let un } cliente_2 = \\
\quad \lambda\textbf{un } acessoClt : \textbf{Request } Cliente. & \quad \lambda\textbf{un } acessoClt : \textbf{Request } Cliente. \\
\quad\quad \textbf{let lin } c_0 = \textbf{request } acessoClt \textbf{ in} & \quad\quad \textbf{let lin } c_0 = \textbf{request } acessoClt \textbf{ in} \\
\quad\quad \textbf{let lin } c_1 = \textbf{send } 0\ c_0 \textbf{ in} & \quad\quad \textbf{let lin } c_1 = \textbf{send } 1\ c_0 \textbf{ in} \\
\quad\quad \textbf{let } \{b, c_2\} = \textbf{receive } c_1 \textbf{ in} & \quad\quad \textbf{let } \{b, c_2\} = \textbf{receive } c_1 \textbf{ in} \\
\quad\quad \textbf{let } \{\} = \textbf{close } c_2 \textbf{ in} & \quad\quad \textbf{let } \{\} = \textbf{close } c_2 \textbf{ in} \\
\quad\quad fazAlgoComBool\ b & \quad\quad b \\
\textbf{in} & \textbf{in } \ldots
\end{array}
$$

O $cliente_1$ rodará em uma *thread* filha, então é necessário que ele retorne algo do tipo $\mathbf{1}$, em vez de $Bool$. Mas como a variável b é linear, ela não pode ser descartada sem ser usada. Assumimos então que a função $fazAlgoComBool$ usa b (por exemplo, enviando-o através de outro canal) e devolve algo do tipo $\mathbf{1}$. Por fim, executamos as *threads*:

$$
\begin{array}{l}
\textbf{new access } [Servidor]\ acessoSrv, acessoClt \textbf{ in} \\
\textbf{let } \{\} = \textbf{spawn } (loopServidor\ acessoSrv) \textbf{ in} \\
\textbf{let } \{\} = \textbf{spawn } (cliente_1\ acessoClt) \textbf{ in} \\
cliente_2\ acessoClt
\end{array}
$$

Como as *threads* dois dois clientes rodam em paralelo e de forma independente (sem sincronizações entre elas), a ordem com que elas se conectarão ao servidor é não-determinística: a *thread* que executar **request** primeiro se conecta ao servidor primeiro. Porém, como o *loop* de servidor sempre gera uma réplica idêntica da implementação de servidor para atender cada cliente, o resultado do programa será sempre o mesmo: $false$. Ou seja, apesar da condição de corrida, o resultado observado ainda é determinístico. Veremos no próximo exemplo que esse nem sempre é o caso. Observe que o programa termina com a *thread* de *loop* do servidor bloqueada em um **accept**, já que não há mais nenhum cliente para ser atendido.

2.7 Estado compartilhado e não-determinismo

Agora, vamos considerar outro exemplo: um servidor de estado compartilhado. O servidor mantém uma variável booleana de estado, cujo valor pode ser lido ou modificado

pelos clientes. Ele oferece duas operações: *get* (opção **left**), em que o servidor envia para o cliente o valor booleano atual do estado; e *put* (opção **right**), em que o cliente envia para o servidor um novo valor booleano para substituir o valor de estado atual. O protocolo é especificado como:

$$Cliente = \oplus\{?Bool.\mathbf{End},\ !Bool.\mathbf{End}\}$$
$$Servidor = \&\{!Bool.\mathbf{End},\ ?Bool.\mathbf{End}\}$$

Quando um cliente quer realizar uma das operações, ele solicita uma conexão com o servidor a partir de um ponto de acesso. O servidor pode ser implementado da seguinte forma:

$$\begin{array}{l}
\mathbf{let\ rec}\ servidor : Bool \multimap (\mathbf{Accept}\ Servidor) \multimap \mathbf{1} = \\
\quad \lambda\mathbf{lin}\ valor : Bool. \\
\quad \lambda\mathbf{un}\ acessoSrv : \mathbf{Accept}\ Servidor. \\
\quad\quad \mathbf{let\ lin}\ s_0 = \mathbf{accept}\ acessoSrv\ \mathbf{in} \\
\quad\quad \mathbf{branch}\ s_0\ \mathbf{of} \\
\quad\quad\quad \mathbf{left}\ s_1 \mapsto \\
\quad\quad\quad\quad \mathbf{let}\ \{valor, valor'\} = duplicateBool\ valor\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let\ lin}\ s_2 = \mathbf{send}\ valor\ s_1\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let}\ \{\} = \mathbf{close}\ s_2\ \mathbf{in} \\
\quad\quad\quad\quad servidor\ valor'\ acessoSrv\ ; \\
\\
\quad\quad\quad \mathbf{right}\ s_1 \mapsto \\
\quad\quad\quad\quad \mathbf{let}\ \{novoValor, s_2\} = \mathbf{receive}\ s_1\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let}\ \{\} = \mathbf{close}\ s_2\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let}\ \{\} = discardBool\ valor\ \mathbf{in} \\
\quad\quad\quad\quad servidor\ novoValor\ acessoSrv \\
\mathbf{in}\ \ldots
\end{array}$$

O servidor fica em um *loop* infinito, atendendo um cliente por vez, de forma sequencial. Como a variável $valor$ é linear, ela deve ser usada exatamente uma vez, e por isso as funções auxiliares $duplicateBool$ e $discardBool$ são usadas para poder duplicar ou descartar o valor. Veremos na Seção 2.8 como essas funções podem ser definidas, e formas alternativas de lidar com valores de forma irrestrita.

Agora, definimos dois clientes, um que realiza o *put* do valor $true$ e outro que realiza um *get* e retorna o valor lido:

$$
\begin{array}{l}
\textbf{let un } clientePut = \\
\quad \lambda \textbf{un } acessoClt : \textbf{Request } Cliente. \\
\qquad \textbf{let lin } c_0 = \textbf{request } acessoClt \textbf{ in} \\
\qquad \textbf{let lin } c_1 = \textbf{select right } c_0 \textbf{ in} \\
\qquad \textbf{let lin } c_2 = \textbf{send } true\ c_1 \textbf{ in} \\
\qquad \textbf{close } c_2 \\
\textbf{in}
\end{array}
\qquad
\begin{array}{l}
\textbf{let un } clienteGet = \\
\quad \lambda \textbf{un } acessoClt : \textbf{Request } Cliente. \\
\qquad \textbf{let lin } c_0 = \textbf{request } acessoClt \textbf{ in} \\
\qquad \textbf{let lin } c_1 = \textbf{select left } c_0 \textbf{ in} \\
\qquad \textbf{let } \{b, c_2\} = \textbf{receive } c_1 \textbf{ in} \\
\qquad \textbf{let } \{\} = \textbf{close } c_2 \textbf{ in} \\
\qquad b \\
\textbf{in } \ldots
\end{array}
$$

Por fim, executamos o servidor e os clientes, conectando-os a partir de um mesmo ponto de acesso:

$$
\begin{array}{l}
\textbf{new access } [Servidor]\ acessoSrv, acessoClt \textbf{ in} \\
\textbf{let } \{\} = \textbf{spawn } (servidor\ false\ acessoSrv) \textbf{ in} \\
\textbf{let } \{\} = \textbf{spawn } (clientePut\ acessoClt) \textbf{ in} \\
clienteGet\ acessoClt
\end{array}
$$

O servidor é inicializado com o valor $false$. O retorno do programa será o valor lido pelo $clienteGet$. Qual será esse valor? Isso dependerá da ordem em que os clientes se conectam com o servidor. Se o $clienteGet$ realiza um **request** primeiro, o valor lido será o inicial, $false$. Por outro lado, se o $clientePut$ realiza um **request** primeiro, o valor lido pelo $clienteGet$ será o $true$ escrito pelo $clientePut$. Portanto, o programa é não-determinístico, ou seja, o resultado observado pode ser diferente a cada execução do programa, dependendo da velocidade de execução de cada thread. Uma situação como essa pode se caracterizar em um *bug* se um dos comportamentos possíveis do programa não é o esperado.

2.8 Estruturas de dados

Nos exemplos anteriores, assumimos os tipos Nat e $Bool$, assim como operadores e sintaxe de literais para esses tipos, como primitivas da linguagem. Por simplicidade, o núcleo de linguagem desenvolvido neste trabalho não inclui essas primitivas, mas possui construções genéricas para definir estruturas de dados. Uma implementação de linguagem mais realista baseada neste núcleo poderia incluir estes tipos como primitivos, tanto por questões de eficiência quanto de conveniência para o programador. Nesta seção, veremos como definir os tipos $Bool$ e Nat, e algumas operações associadas a eles, mostrando que o

núcleo definido aqui é tão poderoso quanto uma versão em que esses tipos são primitivos.

O tipo $Bool$ é definido como:

$$Bool = \mathbf{1} + \mathbf{1}$$

Um tipo $A + B$ é a soma (ou união disjunta) dos tipos A e B. Construímos valores desse tipo usando os construtores $\mathbf{inl}$, que recebe algo do tipo A, e $\mathbf{inr}$, que recebe algo do tipo B. Como o tipo $\mathbf{1}$ possui apenas o valor $\{\}$, o tipo $Bool$ acima possui dois valores, sendo que usaremos o da esquerda (introduzido com $\mathbf{inl}$) para representar $false$ e o da direita (introduzido com $\mathbf{inr}$) para representar $true$:

$$\begin{array}{l}\mathbf{let\ un}\ false = \mathbf{inl}\ [1]\ \{\}\ \mathbf{in} \\ \mathbf{let\ un}\ true = \mathbf{inr}\ [1]\ \{\}\ \mathbf{in}\ \ldots\end{array}$$

No exemplo, é preciso anotar os construtores com o outro tipo da união disjunta (o tipo da direita no $\mathbf{inl}$ e o da esquerda no $\mathbf{inr}$), para que seja possível inferir o tipo da expressão.

Podemos testar se um booleano é $false$ ou $true$ usando uma expressão $\mathbf{case}$. Usaremos isso para definir uma função or:

$$\begin{array}{l}\mathbf{let\ un}\ or = \\ \quad \lambda\mathbf{lin}\ a : Bool. \\ \quad \lambda\mathbf{lin}\ b : Bool. \\ \qquad \mathbf{case}\ a\ \mathbf{of} \\ \qquad\quad \mathbf{inl}\ u \mapsto \\ \qquad\qquad \mathbf{let}\ \{\} = u\ \mathbf{in} \\ \qquad\qquad b\ ; \\ \qquad\quad \mathbf{inr}\ u \mapsto \\ \qquad\qquad \mathbf{let}\ \{\} = u\ \mathbf{in} \\ \qquad\qquad \mathbf{let}\ \{\} = discardBool\ b\ \mathbf{in} \\ \qquad\qquad true \\ \mathbf{in}\ \ldots\end{array}$$

Como a variável b é linear, precisamos usar uma função auxiliar $discardBool$ para descartar a variável, retornando $\{\}$. Essa função também usa um $\mathbf{case}$ para consumir o

booleano:

$$
\begin{array}{l}
\textbf{let un}\ discardBool = \\
\quad \lambda\textbf{lin}\ b : Bool. \\
\qquad \textbf{case}\ b\ \textbf{of} \\
\qquad\quad \textbf{inl}\ u \mapsto\ u\ ; \\
\qquad\quad \textbf{inr}\ u \mapsto\ u \\
\textbf{in}\ \ldots
\end{array}
$$

De forma semelhante, também poderíamos definir a função $duplicateBool$, usada no exemplo do servidor de estado compartilhado da Seção 2.7.

Alternativamente, poderíamos definir o tipo booleano desta forma:

$$Bool' = @(\mathbf{1} + \mathbf{1})$$

Um valor de um tipo $@A$ é uma *closure* de uma expressão do tipo A, cujo valor pode ser recomputado zero ou mais vezes. A notação mais comum para esse tipo na lógica linear é $!A$ (*"of course" A*), mas ela entraria em conflito com a notação do tipo $!A.B$. Um valor desse tipo é construído aplicando o construtor $@$ a uma expressão do tipo A. A expressão não pode conter variáveis lineares livres, já que ela será transformada em uma *closure* que poderá ser descartada ou recomputada diversas vezes. Essa regra também vale para definições irrestritas ("$\textbf{let un}$"), recursivas ("$\textbf{let rec}$") e argumentos de funções irrestritas (funções introduzidas via $\lambda\textbf{un}$). Podemos então definir $false'$ e $true'$ assim:

$$
\begin{array}{l}
\textbf{let un}\ false' = @(\textbf{inl}\ [\mathbf{1}]\ \{\})\ \textbf{in} \\
\textbf{let un}\ true' = @(\textbf{inr}\ [\mathbf{1}]\ \{\})\ \textbf{in}\ \ldots
\end{array}
$$

Podemos agora definir a função $discardBool'$ assim:

$$
\begin{array}{l}
\textbf{let un}\ discardBool' = \\
\quad \lambda\textbf{lin}\ b : Bool. \\
\qquad \textbf{let}\ @x = b\ \textbf{in} \\
\qquad \{\} \\
\textbf{in}\ \ldots
\end{array}
$$

Uma expressão "$\textbf{let}\ @x\ =\ \ldots\ \textbf{in}\ \ldots$" introduz a variável irrestrita x, do tipo A, a qual é vinculada à *closure* guardada por um valor do tipo $@A$. Como a variável é irrestrita, ela pode ser usada zero ou mais vezes (no exemplo, zero), e cada vez em que é usada, a *closure* é recomputada. É necessário recomputar a *closure*, pois ela pode ter

efeitos colaterais. Por exemplo, se a expressão da *closure* for um **fork**, cada vez que ela for recomputada, uma nova *thread* será disparada, um novo canal criado e uma nova extremidade de canal retornada. Se a expressão fosse computada uma única vez e o valor de retorno fosse simplesmente copiado a cada uso da variável irrestrita x, estaríamos usando o mesmo canal de retorno do **fork** zero ou mais vezes, o que seria um problema, como vimos na Seção 2.2 sobre linearidade.

Agora, definimos o tipo Nat. Um natural é ou zero ou o sucessor de outro natural:

$$Nat = \mathbf{rec}\ X : *_{NS}.\ \mathbf{1} + X$$

Note que a variável recursiva é anotada com $*_{NS}$ (*Non-Session*).

Os construtores de naturais $zero$ e $succ$ são definidos da seguinte maneira:

$$\begin{array}{l}
\mathbf{let\ un}\ zero = \mathbf{fold}\ [Nat]\ (\mathbf{inl}\ [Nat]\ \{\})\ \mathbf{in} \\
\\
\mathbf{let\ un}\ succ = \\
\quad \lambda\mathbf{lin}\ n : Nat. \\
\quad\quad \mathbf{fold}\ [Nat]\ (\mathbf{inr}\ [\mathbf{1}]\ n) \\
\mathbf{in}\ \ldots
\end{array}$$

A operação de **fold** faz o inverso de **unfold**: ela recebe algo do tipo $A[X := \mathbf{rec}\ X.A]$ e devolve algo do tipo $\mathbf{rec}\ X.A$. Ela precisa receber como parâmetro o tipo recursivo que está sendo introduzido.

Podemos agora definir uma função add:

$$\begin{array}{l}
\mathbf{let\ rec}\ add : Nat \multimap Nat \multimap Nat = \\
\quad \lambda\mathbf{lin}\ n : Nat. \\
\quad \lambda\mathbf{lin}\ m : Nat. \\
\quad\quad \mathbf{case\ unfold}\ n\ \mathbf{of} \\
\quad\quad\quad \mathbf{inl}\ u \mapsto \\
\quad\quad\quad\quad \mathbf{let}\ \{\} = u\ \mathbf{in} \\
\quad\quad\quad\quad m\ ; \\
\\
\quad\quad\quad \mathbf{inr}\ pred_n \mapsto \\
\quad\quad\quad\quad succ\ (add\ pred_n\ m) \\
\mathbf{in}\ \ldots
\end{array}$$

De forma semelhante, poderíamos definir outras funções, como teste de zero.

3 ESPECIFICAÇÃO FORMAL DA LINGUAGEM

Neste capítulo, serão definidas de forma mais precisa a sintaxe, o sistema de tipos e a semântica operacional da linguagem.

3.1 Sintaxe

A Figura 3.1 apresenta a sintaxe de *kinds*, tipos e termos da linguagem. Os *kinds* são usados para classificar os tipos em tipos de sessão ($*_S$) ou tipos que não são de sessão (tipos de dados ou de ponto de acesso; $*_{NS}$, de *Non-Session*). Na sintaxe de termos, os nomes de canais, c, aparecem apenas em passos intermediários da computação e não estão disponíveis na sintaxe usada pelo programador. O restante da sintaxe é o mesmo já apresentado no Capítulo 2.

Figura 3.1: Sintaxe de *kinds*, tipos e termos

$$
\begin{array}{lrcl}
\text{Kinds} & K & ::= & *_S \mid *_{NS} \\
\text{Tipos} & A, B, C & ::= & X \mid \overline{X} \mid \mathbf{rec}\ X : K.A \\
 & & \mid & !A.B \mid ?A.B \mid \oplus\{A,\ B\} \mid \&\{A,\ B\} \mid \mathbf{End} \\
 & & \mid & A \multimap B \mid A \to B \mid A \times B \mid \mathbf{1} \mid A + B \mid @A \\
 & & \mid & \mathbf{Accept}\ A \mid \mathbf{Request}\ A \\
\text{Termos} & M, N, P & ::= & c \mid x \mid \lambda\mathbf{lin}\ x : A.M \mid \lambda\mathbf{un}\ x : A.M \mid M\ N \\
 & & \mid & \{M, N\} \mid \mathbf{let}\ \{x, y\} = M\ \mathbf{in}\ N \mid \{\} \mid \mathbf{let}\ \{\} = M\ \mathbf{in}\ N \\
 & & \mid & \mathbf{inl}\ [B]\ M \mid \mathbf{inr}\ [A]\ N \mid \mathbf{case}\ M\ \mathbf{of\ inl}\ x \mapsto N;\ \mathbf{inr}\ y \mapsto P \\
 & & \mid & @M \mid \mathbf{let}\ @x = M\ \mathbf{in}\ N \\
 & & \mid & \mathbf{send}\ M\ N \mid \mathbf{receive}\ M \mid \mathbf{select\ left}\ M \mid \mathbf{select\ right}\ M \\
 & & \mid & \mathbf{branch}\ M\ \mathbf{of\ left}\ x \mapsto N;\ \mathbf{right}\ y \mapsto P \mid \mathbf{close}\ M \\
 & & \mid & \mathbf{new\ access}\ [A]\ x, y\ \mathbf{in}\ M \mid \mathbf{accept}\ M \mid \mathbf{request}\ M \\
 & & \mid & \mathbf{new\ session}\ [A] \mid \mathbf{fork}\ M \mid \mathbf{spawn}\ M \\
 & & \mid & \mathbf{fold}\ [A]\ M \mid \mathbf{unfold}\ M \\
 & & \mid & \mathbf{let\ lin}\ x = M\ \mathbf{in}\ N \mid \mathbf{let\ un}\ x = M\ \mathbf{in}\ N \\
 & & \mid & \mathbf{let\ rec}\ x : A = M\ \mathbf{in}\ N
\end{array}
$$

Fonte: O Autor

3.2 Sistema de tipos

A Figura 3.2 define os contextos de variáveis usados no sistema de tipos. Um julgamento de tipo faz uso de três contextos: um contexto Θ de variáveis de tipo (introdu-

zidas via definições de tipo ou tipos recursivos $\mathbf{rec}\ X : K.A$), associando a cada variável o seu *kind*; e os contextos Γ, de variáveis de termo irrestritas, e Δ, de variáveis de termo lineares, associando a cada variável de termo o seu tipo. A ordem das declarações de variáveis é irrelevante e sempre assume-se implicitamente que não há variáveis repetidas em um contexto.

Figura 3.2: Contextos de variáveis

Contextos de variáveis de tipo	Θ	$::=$	$\cdot \mid \Theta, X : K$
Contextos irrestritos	Γ	$::=$	$\cdot \mid \Gamma, x : A$
Contextos lineares	Δ	$::=$	$\cdot \mid \Delta, x : A$

Fonte: O Autor

A Figura 3.3 mostra as regras de boa-formação de contextos e tipos. O julgamento $\Theta; \Gamma; \Delta \vdash \mathbf{ok}$ afirma que os contextos Θ, Γ e Δ, em combinação, são bem-formados. Em outras palavras, cada tipo que ocorre no contexto irrestrito Γ ou linear Δ é bem-formado em relação ao contexto de variáveis de tipo Θ. Já o julgamento $\Theta \vdash A : K$ afirma que o tipo A é bem-formado e do *kind* K no contexto de variáveis de tipo Θ. Isso significa que todas as variáveis de tipo que ocorrem em A são declaradas em Θ. Também significa que A não é um tipo mal-formado como, por exemplo, $!B.\mathbf{1}$, onde o tipo da continuação, $\mathbf{1}$, não é um tipo de sessão.

A dualidade $\overline{A}$, definida na Figura 3.4, é uma operação parcial sobre tipos, e é definida se e somente se A é um tipo de sessão, ou seja, se $\Theta \vdash A : *_S$ para algum Θ. O dual de uma variável de tipo, $\overline{X}$, faz parte da sintaxe.

A operação de substituição em tipos, $A[X := B]$, substitui, em A, todas as ocorrências livres de X por B e todas as ocorrências livres de $\overline{X}$ por $\overline{B}$.

A definição do dual de um tipo recursivo é um pouco complicada, pois envolve a substituição de X por $\overline{X}$ no corpo do tipo recursivo. Para entender a necessidade dessa substituição, considere dois exemplos.

Primeiro, o tipo recursivo $\mathbf{rec}\ X.!\mathbf{Nat}.X$ (anotação de *kind* omitida), que define um protocolo que repetidamente envia $\mathbf{Nat}$s. Se tomássemos o dual desse tipo sem substituir X por $\overline{X}$, teríamos o tipo $\mathbf{rec}\ X.?\mathbf{Nat}.\overline{X}$, que definiria um protocolo que primeiro recebe um $\mathbf{Nat}$, depois envia $\mathbf{Nat}$s repetidamente, e claramente não é o dual correto. Nesse caso, a ocorrência de X não deveria ser dualizada, e o dual correto seria $\mathbf{rec}\ X.?\mathbf{Nat}.X$ (o X é substituído por $\overline{X}$ e depois dualizado, voltando a ser X).

O segundo exemplo é um tipo $\mathbf{Cliente} := \mathbf{rec}\ X.!X.\mathbf{End}$. O cliente envia algo do tipo X e termina. Mas a ocorrência de X faz referência ao próprio tipo recursivo

Figura 3.3: Regras de boa-formação de contextos e tipos

Boa-formação de contextos

$$\frac{}{\cdot;\cdot;\cdot \vdash \mathbf{ok}}\ \text{EmptyOK} \qquad \frac{\Theta;\Gamma;\Delta \vdash \mathbf{ok}}{(\Theta, X : K);\Gamma;\Delta \vdash \mathbf{ok}}\ \text{TypeVarOK}$$

$$\frac{\Theta;\Gamma;\Delta \vdash \mathbf{ok} \qquad \Theta \vdash A : K}{\Theta;(\Gamma, x : A);\Delta \vdash \mathbf{ok}}\ \text{UnVarOK} \qquad \frac{\Theta;\Gamma;\Delta \vdash \mathbf{ok} \qquad \Theta \vdash A : K}{\Theta;\Gamma;(\Delta, x : A) \vdash \mathbf{ok}}\ \text{LinVarOK}$$

Boa-formação de tipos

$$\frac{}{\Theta, X : K \vdash X : K}\ \text{TypeVarWF} \qquad \frac{}{\Theta, X : *_S \vdash \overline{X} : *_S}\ \text{TypeVarDualWF}$$

$$\frac{\Theta, X : K \vdash A : K}{\Theta \vdash \mathbf{rec}\ X : K.A : K}\ \text{RecWF}$$

$$\frac{\Theta \vdash A : K \qquad \Theta \vdash B : *_S}{\Theta \vdash !A.B : *_S}\ \text{SendWF} \qquad \frac{\Theta \vdash A : K \qquad \Theta \vdash B : *_S}{\Theta \vdash ?A.B : *_S}\ \text{ReceiveWF}$$

$$\frac{\Theta \vdash A : *_S \qquad \Theta \vdash B : *_S}{\Theta \vdash \oplus\{A,\ B\} : *_S}\ \text{SelectWF} \qquad \frac{\Theta \vdash A : *_S \qquad \Theta \vdash B : *_S}{\Theta \vdash \&\{A,\ B\} : *_S}\ \text{BranchWF}$$

$$\frac{}{\Theta \vdash \mathbf{End} : *_S}\ \text{EndWF}$$

$$\frac{\Theta \vdash A : *_S}{\Theta \vdash \mathbf{Accept}\ A : *_{NS}}\ \text{AcceptWF} \qquad \frac{\Theta \vdash A : *_S}{\Theta \vdash \mathbf{Request}\ A : *_{NS}}\ \text{RequestWF}$$

$$\frac{\Theta \vdash A : K \qquad \Theta \vdash B : K'}{\Theta \vdash A \multimap B : *_{NS}}\ \text{LinFunWF} \qquad \frac{\Theta \vdash A : K \qquad \Theta \vdash B : K'}{\Theta \vdash A \to B : *_{NS}}\ \text{UnFunWF}$$

$$\frac{\Theta \vdash A : K \qquad \Theta \vdash B : K'}{\Theta \vdash A \times B : *_{NS}}\ \text{ProductWF} \qquad \frac{}{\Theta \vdash \mathbf{1} : *_{NS}}\ \text{OneWF}$$

$$\frac{\Theta \vdash A : K \qquad \Theta \vdash B : K'}{\Theta \vdash A + B : *_{NS}}\ \text{SumWF} \qquad \frac{\Theta \vdash A : K}{\Theta \vdash @A : *_{NS}}\ \text{OfCourseWF}$$

Fonte: O Autor

Figura 3.4: Dualidade de tipos de sessão

$$\overline{!A.B} = ?A.\overline{B} \qquad \overline{?A.B} = !A.\overline{B}$$

$$\overline{\oplus\{A,\ B\}} = \&\{\overline{A},\ \overline{B}\} \qquad \overline{\&\{A,\ B\}} = \oplus\{\overline{A},\ \overline{B}\}$$

$$\overline{\mathbf{End}} = \mathbf{End}$$

$$\overline{(X)} = \overline{X} \qquad \overline{(\overline{X})} = X$$

$$\overline{\mathbf{rec}\ X : K.A} = \mathbf{rec}\ X : K.\overline{A[X := \overline{X}]}$$

Fonte: O Autor

$\mathbf{Cliente}$, portanto o valor enviado é um canal do tipo $\mathbf{Cliente}$. Se definirmos um servidor como sendo o dual sem a substituição de variável, temos $\mathbf{Servidor} := \mathbf{rec}\ X.\ ?X.\mathbf{End}$. O servidor espera receber algo do tipo X, mas agora X faz referência ao tipo recursivo $\mathbf{Servidor}$, portanto o servidor espera um valor de um tipo diferente daquele enviado pelo cliente. Nesse caso, o que ocorre é o oposto do primeiro exemplo: a ocorrência de X não foi dualizada, mas deveria, e o dual correto seria $\mathbf{rec}\ X.\ ?\overline{X}.\mathbf{End}$ (o X é substituído por $\overline{X}$).

As Figuras 3.5 e 3.6 mostram as regras do sistema de tipos da linguagem. O julgamento de tipo $\Theta; \Gamma; \Delta \vdash M : A$ afirma que, sob os contextos de variáveis de tipos Θ, variáveis irrestritas Γ e variáveis lineares Δ, o termo M é bem-tipado e do tipo A (bem-formado). Assume-se implicitamente que os contextos são bem-formados ($\Theta; \Gamma; \Delta \vdash \mathbf{ok}$). As regras de tipo garantem que os contextos referenciados nas premissas também são bem-formados. Por exemplo, na regra $\multimap$-I, a primeira premissa garante que o tipo anotado A é bem-formado e, portanto, a extensão do contexto na segunda premissa é bem-formada.

As restrições de linearidade da linguagem são asseguradas pela forma como os contextos são manipulados pelas regras de tipos. Os contextos de variáveis irrestritas e de tipo são manipulados da mesma forma que em uma linguagem funcional convencional (sem linearidade), permitindo que variáveis sejam duplicadas ou descartadas livremente. Já o contexto linear é manipulado de forma mais rigorosa, garantindo que as variáveis não sejam duplicadas, e que as variáveis que ocorrem no contexto sejam apenas aquelas que são usadas no termo sendo tipado (sem descartar variáveis).

A regra LinVar permite que uma variável linear seja usada. O contexto linear deve conter apenas a variável x que está sendo usada, pois se o contexto tivesse outras variáveis lineares, elas estariam sendo descartadas sem serem usadas. De forma parecida, a regra UnVar permite que uma variável irrestrita seja usada. Nesse caso, o contexto linear é vazio, pois nenhuma variável linear está sendo usada. O mesmo ocorre nas regras $\mathbf{1}$-I e NewSession, onde nenhuma variável linear é usada.

Nas regras onde uma variável linear deve ser introduzida (por exemplo, a regra $\multimap$-I, de abstração de variável linear), isso é feito estendendo o contexto linear (Δ, na regra $\multimap$-I) com uma variável. De forma semelhante, nas regras onde uma variável irrestrita deve ser introduzida (por exemplo, a regra $\to$-I, de abstração de variável irrestrita), o contexto irrestrito (Γ) é estendido.

Na maior parte das regras, como por exemplo $\times$-I, os contextos Θ e Γ são dupli-

Figura 3.5: Regras de tipo para a parte puramente funcional da linguagem

$$\frac{}{\Theta; \Gamma; x : A \vdash x : A}\ \text{LinVar} \qquad \frac{}{\Theta; (\Gamma, x : A); \cdot \vdash x : A}\ \text{UnVar}$$

$$\frac{\Theta \vdash A : K \qquad \Theta; \Gamma; (\Delta, x : A) \vdash M : B}{\Theta; \Gamma; \Delta \vdash \lambda\mathbf{lin}\ x : A.M : A \multimap B}\ \multimap\text{-I}$$

$$\frac{\Theta; \Gamma; \Delta \vdash M : A \multimap B \qquad \Theta; \Gamma; \Delta' \vdash N : A}{\Theta; \Gamma; (\Delta, \Delta') \vdash M\ N : B}\ \multimap\text{-E}$$

$$\frac{\Theta \vdash A : K \qquad \Theta; (\Gamma, x : A); \Delta \vdash M : B}{\Theta; \Gamma; \Delta \vdash \lambda\mathbf{un}\ x : A.M : A \to B}\ \to\text{-I}$$

$$\frac{\Theta; \Gamma; \Delta \vdash M : A \to B \qquad \Theta; \Gamma; \cdot \vdash N : A}{\Theta; \Gamma; \Delta \vdash M\ N : B}\ \to\text{-E}$$

$$\frac{\Theta; \Gamma; \Delta \vdash M : A \qquad \Theta; \Gamma; \Delta' \vdash N : B}{\Theta; \Gamma; (\Delta, \Delta') \vdash \{M, N\} : A \times B}\ \times\text{-I}$$

$$\frac{\Theta; \Gamma; \Delta \vdash M : A \times B \qquad \Theta; \Gamma; (\Delta', x : A, y : B) \vdash N : C}{\Theta; \Gamma; (\Delta, \Delta') \vdash \mathbf{let}\ \{x, y\} = M\ \mathbf{in}\ N : C}\ \times\text{-E}$$

$$\frac{}{\Theta; \Gamma; \cdot \vdash \{\} : \mathbf{1}}\ \mathbf{1}\text{-I} \qquad \frac{\Theta; \Gamma; \Delta \vdash M : \mathbf{1} \qquad \Theta; \Gamma; \Delta' \vdash N : C}{\Theta; \Gamma; (\Delta, \Delta') \vdash \mathbf{let}\ \{\} = M\ \mathbf{in}\ N : C}\ \mathbf{1}\text{-E}$$

$$\frac{\Theta \vdash B : K \qquad \Theta; \Gamma; \Delta \vdash M : A}{\Theta; \Gamma; \Delta \vdash \mathbf{inl}\ [B]\ M : A + B}\ +\text{-I}_1 \qquad \frac{\Theta \vdash A : K \qquad \Theta; \Gamma; \Delta \vdash N : B}{\Theta; \Gamma; \Delta \vdash \mathbf{inr}\ [A]\ N : A + B}\ +\text{-I}_2$$

$$\frac{\Theta; \Gamma; \Delta \vdash M : A + B \qquad \Theta; \Gamma; (\Delta', x : A) \vdash N : C \qquad \Theta; \Gamma; (\Delta', y : B) \vdash P : C}{\Theta; \Gamma; (\Delta, \Delta') \vdash \mathbf{case}\ M\ \mathbf{of}\ \mathbf{inl}\ x \mapsto N;\ \mathbf{inr}\ y \mapsto P : C}\ +\text{-E}$$

$$\frac{\Theta; \Gamma; \cdot \vdash M : A}{\Theta; \Gamma; \cdot \vdash @M : @A}\ @\text{-I} \qquad \frac{\Theta; \Gamma; \Delta \vdash M : @A \qquad \Theta; (\Gamma, x : A); \Delta' \vdash N : B}{\Theta; \Gamma; (\Delta, \Delta') \vdash \mathbf{let}\ @x = M\ \mathbf{in}\ N : B}\ @\text{-E}$$

$$\frac{\Theta \vdash \mathbf{rec}\ X : K.A : K' \qquad \Theta; \Gamma; \Delta \vdash M : A[X := \mathbf{rec}\ X : K.A]}{\Theta; \Gamma; \Delta \vdash \mathbf{fold}\ [\mathbf{rec}\ X : K.A]\ M : \mathbf{rec}\ X : K.A}\ \mathbf{rec}\text{-I}$$

$$\frac{\Theta; \Gamma; \Delta \vdash M : \mathbf{rec}\ X : K.A}{\Theta; \Gamma; \Delta \vdash \mathbf{unfold}\ M : A[X := \mathbf{rec}\ X : K.A]}\ \mathbf{rec}\text{-E}$$

$$\frac{\Theta; \Gamma; \Delta \vdash M : A \qquad \Theta; \Gamma; (\Delta', x : A) \vdash N : B}{\Theta; \Gamma; (\Delta, \Delta') \vdash \mathbf{let\ lin}\ x = M\ \mathbf{in}\ N : B}\ \text{LetLin}$$

$$\frac{\Theta; \Gamma; \cdot \vdash M : A \qquad \Theta; (\Gamma, x : A); \Delta \vdash N : B}{\Theta; \Gamma; \Delta \vdash \mathbf{let\ un}\ x = M\ \mathbf{in}\ N : B}\ \text{LetUn}$$

$$\frac{\Theta \vdash A : K \qquad \Theta; (\Gamma, x : A); \cdot \vdash M : A \qquad \Theta; (\Gamma, x : A); \Delta \vdash N : B}{\Theta; \Gamma; \Delta \vdash \mathbf{let\ rec}\ x : A = M\ \mathbf{in}\ N : B}\ \text{LetRec}$$

Fonte: O Autor

Figura 3.6: Regras de tipo para operações de concorrência

$$\frac{\Theta;\Gamma;\Delta \vdash M : A \qquad \Theta;\Gamma;\Delta' \vdash N : !A.B}{\Theta;\Gamma;(\Delta,\Delta') \vdash \mathbf{send}\ M\ N : B}\ \text{Send}$$

$$\frac{\Theta;\Gamma;\Delta \vdash M : ?A.B}{\Theta;\Gamma;\Delta \vdash \mathbf{receive}\ M : A \times B}\ \text{Receive}$$

$$\frac{\Theta;\Gamma;\Delta \vdash M : \oplus\{A,\ B\}}{\Theta;\Gamma;\Delta \vdash \mathbf{select\ left}\ M : A}\ \text{SelectL} \qquad \frac{\Theta;\Gamma;\Delta \vdash M : \oplus\{A,\ B\}}{\Theta;\Gamma;\Delta \vdash \mathbf{select\ right}\ M : B}\ \text{SelectR}$$

$$\frac{\Theta;\Gamma;\Delta \vdash M : \&\{A,\ B\} \qquad \Theta;\Gamma;(\Delta', x : A) \vdash N : C \qquad \Theta;\Gamma;(\Delta', y : B) \vdash P : C}{\Theta;\Gamma;(\Delta,\Delta') \vdash \mathbf{branch}\ M\ \mathbf{of\ left}\ x \mapsto N;\ \mathbf{right}\ y \mapsto P : C}\ \text{Branch}$$

$$\frac{\Theta;\Gamma;\Delta \vdash M : \mathbf{End}}{\Theta;\Gamma;\Delta \vdash \mathbf{close}\ M : \mathbf{1}}\ \text{Close}$$

$$\frac{\Theta \vdash A : *_S \qquad \Theta;(\Gamma, x : \mathbf{Accept}\ A, y : \mathbf{Request}\ \overline{A});\Delta \vdash M : B}{\Theta;\Gamma;\Delta \vdash \mathbf{new\ access}\ [A]\ x, y\ \mathbf{in}\ M : B}\ \text{NewAccess}$$

$$\frac{\Theta;\Gamma;\Delta \vdash M : \mathbf{Accept}\ A}{\Theta;\Gamma;\Delta \vdash \mathbf{accept}\ M : A}\ \text{Accept} \qquad \frac{\Theta;\Gamma;\Delta \vdash M : \mathbf{Request}\ A}{\Theta;\Gamma;\Delta \vdash \mathbf{request}\ M : A}\ \text{Request}$$

$$\frac{\Theta \vdash A : *_S}{\Theta;\Gamma;\cdot \vdash \mathbf{new\ session}\ [A] : A \times \overline{A}}\ \text{NewSession}$$

$$\frac{\Theta;\Gamma;\Delta \vdash M : A \multimap \mathbf{1} \qquad \Theta \vdash A : *_S}{\Theta;\Gamma;\Delta \vdash \mathbf{fork}\ M : \overline{A}}\ \text{Fork}$$

$$\frac{\Theta;\Gamma;\Delta \vdash M : \mathbf{1}}{\Theta;\Gamma;\Delta \vdash \mathbf{spawn}\ M : \mathbf{1}}\ \text{Spawn}$$

Fonte: O Autor

cados para ambas as premissas, enquanto o contexto linear é dividido em duas partes, Δ e Δ', cada uma sendo usada em uma das premissas. Isso garante que nenhuma variável linear é duplicada.

Nas regras $+$-E e Branch, o contexto linear Δ' parece estar sendo duplicado. Porém, isso não acontece, porque apenas um dos ramos será executado. Os contextos lineares dos dois ramos devem ser iguais pois, se uma variável linear ocorresse em um dos ramos, mas não no outro, o sistema de tipos não seria capaz de decidir se a variável seria usada zero ou uma vezes.

Na regra $\rightarrow$-E, de aplicação de função irrestrita, o contexto linear do argumento N, na segunda premissa, é vazio, o que significa que nenhuma variável linear livre deve ser usada no argumento. Como a função aplicada usa o argumento de forma irrestrita, qualquer variável linear ocorrendo em N poderia ser descartada ou duplicada junto com N. O mesmo ocorre nas regras @-I e LetUn. Na regra LetRec, o corpo da definição recursiva, M, pode se desenrolar diversas vezes durante a recursão. Além disso, a definição é usada de forma irrestrita em N. Logo, o contexto linear de M também deve ser vazio.

3.3 Semântica operacional

A semântica operacional da linguagem é apresentada no estilo *small-step*, usando uma abordagem baseada no cálculo-λ para modelar a computação puramente funcional e sequencial em uma *thread*, e operadores baseados no cálculo-π para modelar os aspectos concorrentes da execução do programa. A Figura 3.7 mostra a sintaxe usada na avaliação de programas. Nessa sintaxe, as anotações de tipos são omitidas.

Um valor c corresponde a um nome de canal do cálculo-π, e é usado para modelar tanto canais de sessão quanto pontos de acesso.

Um contexto de avaliação E é um termo com um "buraco", denotado por $[\,]$. O buraco indica a posição, dentro do termo, onde a próxima redução deve ocorrer. A notação $E[M]$ representa o preenchimento do (único) buraco de E com o termo M. Veremos mais adiante que os contextos de avaliação facilitam a definição de reduções envolvendo múltiplas *threads* como, por exemplo, a sincronização entre as operações de $\mathbf{send}$ e $\mathbf{receive}$.

Uma configuração C modela o estado das *threads* e canais criados durante a execução do programa. Uma *thread* ϕM executa um termo M de forma sequencial. A *flag* ϕ indica se a *thread* é a principal ($\bullet$) ou uma *thread* filha ($\circ$), criada com $\mathbf{fork}$ ou $\mathbf{spawn}$. A *thread* principal é aquela que retorna o resultado do programa, e deve ser única em

Figura 3.7: Sintaxe de valores, configurações e contextos de avaliação

Valores	V, W	$::=$	$c \mid \lambda\mathbf{lin}\ x.M \mid \lambda\mathbf{un}\ x.M \mid \{V, W\} \mid \{\}$
		$\mid$	$\mathbf{inl}\ V \mid \mathbf{inr}\ W \mid @M \mid \mathbf{fold}\ V$
Contextos de avaliação	E	$::=$	$[\,] \mid E\ N \mid (\lambda\mathbf{lin}\ x.M)\ E$
		$\mid$	$\{E, N\} \mid \{V, E\} \mid \mathbf{let}\ \{x, y\} = E\ \mathbf{in}\ N$
		$\mid$	$\mathbf{let}\ \{\} = E\ \mathbf{in}\ N \mid \mathbf{inl}\ E \mid \mathbf{inr}\ E$
		$\mid$	$\mathbf{case}\ E\ \mathbf{of\ inl}\ x \mapsto N;\ \mathbf{inr}\ y \mapsto P$
		$\mid$	$\mathbf{let}\ @x = E\ \mathbf{in}\ N \mid \mathbf{send}\ E\ N \mid \mathbf{send}\ V\ E$
		$\mid$	$\mathbf{receive}\ E \mid \mathbf{select\ left}\ E \mid \mathbf{select\ right}\ E$
		$\mid$	$\mathbf{branch}\ E\ \mathbf{of\ left}\ x \mapsto N;\ \mathbf{right}\ y \mapsto P$
		$\mid$	$\mathbf{close}\ E \mid \mathbf{accept}\ E \mid \mathbf{request}\ E \mid \mathbf{fork}\ E$
		$\mid$	$\mathbf{fold}\ E \mid \mathbf{unfold}\ E \mid \mathbf{let\ lin}\ x = E\ \mathbf{in}\ N$
Configurações	C, D	$::=$	$\phi M \mid C \parallel D \mid (\nu c)C$
Contextos de configuração	G	$::=$	$[\,] \mid G \parallel D \mid (\nu c)G$
Flags	ϕ	$::=$	$\bullet \mid \circ$

Fonte: O Autor

uma configuração. Já as *threads* filhas devem sempre retornar $\{\}$. A composição paralela $C \parallel D$ denota a execução das configurações C e D em paralelo. Uma restrição de nome $(\nu c)C$ denota a existência de um canal c compartilhado entre as *threads* de C. Os operadores de composição paralela e restrição de nome são originários do cálculo-π. De forma semelhante aos contextos de avaliação, um contexto de configuração G é uma configuração com um "buraco", indicando onde uma redução pode ocorrer.

A semântica da linguagem é dividida em uma relação de redução determinística sobre termos ($\longrightarrow_M$) e uma relação de redução não-determinística sobre configurações ($\longrightarrow$). A Figura 3.8 define a relação de redução sobre termos. A última regra estabelece que uma redução de termo pode ocorrer dentro de um contexto de avaliação E.

Expressões vinculadas a variáveis irrestritas seguem uma estratégia de avaliação *call-by-name*, onde a expressão não é avaliada imediatamente, mas apenas se e quando seu valor é necessário, e reavaliada toda vez em que esse valor é necessário novamente. Como foi apontado na Seção 2.8, reavaliar a expressão é importante, pois ela pode conter efeitos colaterais que precisam ser reexecutados, produzindo um novo valor a cada execução.

Por outro lado, expressões vinculadas a variáveis lineares são avaliadas seguindo uma estratégia *call-by-value*, onde a expressão é imediatamente avaliada para um valor. Essa estratégia é interessante já que, se o programa for bem-tipado, temos a garantia de que o valor será necessário *exatamente uma vez*, logo a computação não será em vão, e também não ocorrerão problemas relacionados à reavaliação de efeitos colaterais.

Esse estilo de semântica operacional, que combina as estratégias *call-by-value*

Figura 3.8: Regras de redução de termos

$$
\begin{array}{rcl}
(\lambda \mathbf{lin}\ x.M)\ V & \longrightarrow_M & M[x := V] \\
(\lambda \mathbf{un}\ x.M)\ N & \longrightarrow_M & M[x := N] \\
\mathbf{let}\ \{x, y\} = \{V, W\}\ \mathbf{in}\ N & \longrightarrow_M & N[x := V, y := W] \\
\mathbf{let}\ \{\} = \{\}\ \mathbf{in}\ N & \longrightarrow_M & N \\
\mathbf{case\ inl}\ V\ \mathbf{of\ inl}\ x \mapsto N;\ \mathbf{inr}\ y \mapsto P & \longrightarrow_M & N[x := V] \\
\mathbf{case\ inr}\ W\ \mathbf{of\ inl}\ x \mapsto N;\ \mathbf{inr}\ y \mapsto P & \longrightarrow_M & P[y := W] \\
\mathbf{let}\ @x = @M\ \mathbf{in}\ N & \longrightarrow_M & N[x := M] \\
\mathbf{unfold}\ (\mathbf{fold}\ V) & \longrightarrow_M & V \\
\mathbf{unfold}\ c & \longrightarrow_M & c \\
\mathbf{let\ lin}\ x = V\ \mathbf{in}\ N & \longrightarrow_M & N[x := V] \\
\mathbf{let\ un}\ x = M\ \mathbf{in}\ N & \longrightarrow_M & N[x := M] \\
\mathbf{let\ rec}\ x = M\ \mathbf{in}\ N & \longrightarrow_M & N[x := \mathbf{let\ rec}\ x = M\ \mathbf{in}\ M]
\end{array}
$$

$$
\frac{M \longrightarrow_M M'}{E[M] \longrightarrow_M E[M']}
$$

Fonte: O Autor

e *call-by-name*, é comum na literatura de lógica e linguagens de programação lineares, podendo ser vista, por exemplo, em Abramsky (1993) e Turner e Wadler (1999).

A Figura 3.9 apresenta as regras de equivalência e redução de configurações. A relação $\equiv$, de equivalência estrutural de configurações, é a menor relação de equivalência satisfazendo as regras apresentadas. Essa equivalência estabelece que certas diferenças estruturais, como a ordem das *threads* sob composição paralela e a ordem das declarações de canais, são irrelevantes, e permite que uma configuração seja rearranjada para que as regras de redução possam ser aplicadas. Escreve-se $fc(C)$ para denotar o conjunto de nomes de canal livres em C.

Nas regras de redução de configurações é possível ver a necessidade dos contextos de avaliação. Termos como $E[\mathbf{send}\ V\ c]$ e $E'[\mathbf{receive}\ c]$, na primeira regra, estão bloqueados em operações de concorrência. Observe que eles não podem ser reduzidos pelas regras de redução de termos, sendo de fato termos presos sob a relação $\longrightarrow_M$. Com os contextos de avaliação, é possível expressar que essas operações de **send** e **receive**, sobre o mesmo canal c, podem ser reduzidas simultaneamente, dentro de um contexto onde suas *threads* estão compostas em paralelo e sob o escopo do canal c.

A última regra de redução define que se uma configuração C' reduz para D', então qualquer configuração C equivalente a C' reduz para qualquer configuração D equivalente a D'. Em outras palavras, a redução de configurações ignora diferenças meramente estruturais, conforme estabelecido pela relação $\equiv$.

Figura 3.9: Regras de equivalência e redução para configurações

Equivalência estrutural de configurações

$$C \parallel D \equiv D \parallel C \qquad C \parallel (D \parallel E) \equiv (C \parallel D) \parallel E \qquad C \parallel \circ\{\} \equiv C$$

$$(\nu c)(\nu d)C \equiv (\nu d)(\nu c)C \qquad C \parallel (\nu c)D \equiv (\nu c)(C \parallel D) \quad \text{se } c \notin fc(C)$$

$$\frac{C \equiv D}{G[C] \equiv G[D]}$$

Redução de configurações

$$(\nu c)(\phi E[\mathbf{send}\ V\ c] \parallel \phi' E'[\mathbf{receive}\ c]) \longrightarrow (\nu c)(\phi E[c] \parallel \phi' E'[\{V, c\}])$$

$$(\nu c)(\phi E[\mathbf{select\ left}\ c] \parallel \phi' E'[\mathbf{branch}\ c\ \mathbf{of\ left}\ x \mapsto N;\ \mathbf{right}\ y \mapsto P]) \longrightarrow (\nu c)(\phi E[c] \parallel \phi' E'[N[x := c]])$$

$$(\nu c)(\phi E[\mathbf{select\ right}\ c] \parallel \phi' E'[\mathbf{branch}\ c\ \mathbf{of\ left}\ x \mapsto N;\ \mathbf{right}\ y \mapsto P]) \longrightarrow (\nu c)(\phi E[c] \parallel \phi' E'[P[y := c]])$$

$$(\nu c)(\phi E[\mathbf{close}\ c] \parallel \phi' E'[\mathbf{close}\ c]) \longrightarrow \phi E[\{\}] \parallel \phi' E'[\{\}]$$

$$\phi E[\mathbf{new\ access}\ x, y\ \mathbf{in}\ M] \longrightarrow (\nu c)(\phi E[M[x := c][y := c]]) \qquad c \text{ novo}$$

$$(\nu c)(\phi E[\mathbf{accept}\ c] \parallel \phi' E'[\mathbf{request}\ c]) \longrightarrow (\nu c)(\nu d)(\phi E[d] \parallel \phi' E'[d]) \qquad d \text{ novo}$$

$$\phi E[\mathbf{new\ session}] \longrightarrow (\nu c)(\phi E[\{c, c\}]) \qquad c \text{ novo}$$

$$\phi E[\mathbf{fork}\ (\lambda \mathbf{lin}\ x.M)] \longrightarrow (\nu c)(\phi E[c] \parallel \circ M[x := c]) \qquad c \text{ novo}$$

$$\phi E[\mathbf{spawn}\ M] \longrightarrow \phi E[\{\}] \parallel \circ M$$

$$\frac{M \longrightarrow_M M'}{\phi M \longrightarrow \phi M'} \qquad \frac{C \longrightarrow C'}{G[C] \longrightarrow G[C']}$$

$$\frac{C \equiv C' \qquad C' \longrightarrow D' \qquad D' \equiv D}{C \longrightarrow D}$$

Fonte: O Autor

4 INTERPRETADOR

Este capítulo descreve a implementação de um protótipo de interpretador baseada na especificação formal da linguagem, e como ele pode ser usado.

4.1 Uso do interpretador

A aplicação *web* do interpretador está disponível *on-line*:

<https://gabrieldesh.github.io/concurrent-lambda-calculus>

A Figura 4.1 mostra a interface do interpretador. Os exemplos apresentados no Capítulo 2 também estão disponíveis *on-line*, no repositório da implementação:

<https://github.com/gabrieldesh/concurrent-lambda-calculus/tree/main/examples>

Cada arquivo inclui as definições de Nat e $Bool$ necessárias para rodar o exemplo.

A sintaxe do interpretador substitui os símbolos matemáticos da sintaxe definida na Figura 3.1, do capítulo anterior, por caracteres ASCII, mais convenientes de serem introduzidos pelo teclado. A relação entre a sintaxe matemática e a do interpretador é mostrada na Tabela 4.1.

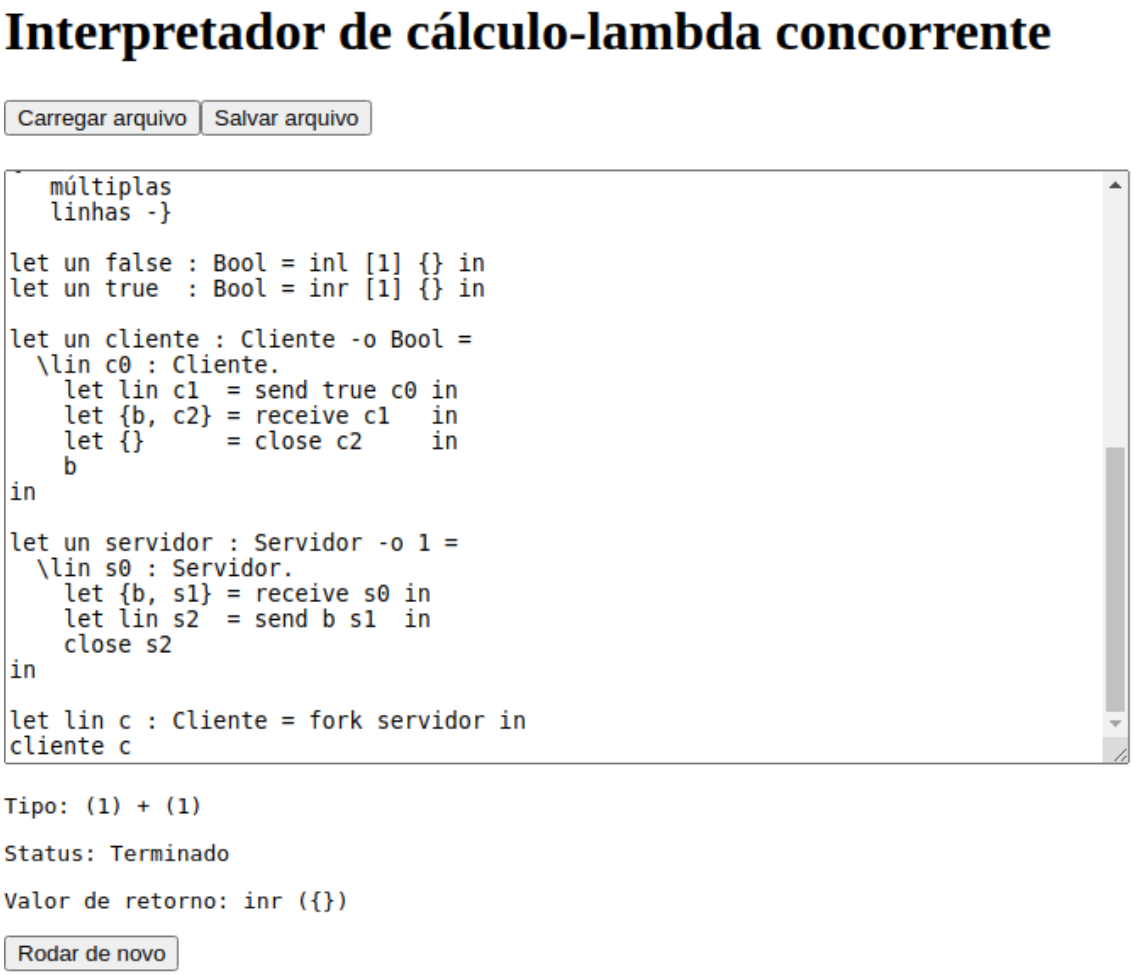
Tabela 4.1: Relação entre a sintaxe formal e a do interpretador

Sintaxe formal	*Sintaxe do interpretador*
$*_S$	`*s`
$*_{NS}$	`*ns`
$\overline{A}$	`dual(A)`
$\oplus$	`+`
$\multimap$	`-o`
$\to$	`->`
$\times$	`*`
λ	`\`
$\mapsto$	`->`

Fonte: O Autor

A Figura 4.2 mostra um exemplo de programa com a sintaxe do interpretador. Um programa começa com três seções opcionais. A seção `typevars` permite declarar os *kinds* de variáveis de tipo livres (sem definição) do programa. A seção `typedefs` permite introduzir definições de tipo, como por exemplo as definições de $Cliente$, $Servidor$, Nat e $Bool$ do Capítulo 2. Essa seção pode fazer uso das variáveis de tipo declaradas na seção `typevars`. A seção `vars` permite declarar os tipos de variáveis de termo lineares

Figura 4.1: Captura de tela mostrando a interface do interpretador



Fonte: O Autor

ou irrestritas livres (sem definição). Essa seção pode fazer uso das variáveis de tipo introduzidas nas seções anteriores. Por fim, temos o termo principal, que é executado pelo interpretador. As variáveis de tipo definidas em `typedefs` são substituidas pelas suas definições dentro da seção `vars` e do termo principal, na ordem em que são definidas. Os itens de cada uma dessas seções são separados por `;`.

O interpretador oferece a operação `dual`, que computa o dual de um tipo, para que o programador não precise inseri-lo manualmente. No exemplo, o tipo de sessão `Cliente` é definido explicitamente, enquanto o `Servidor` é definido como dual de `Cliente`.

As expressões `let lin` e `let un` possuem uma anotação de tipo opcional (a anotação de `let rec` é obrigatória). Quando essa declaração é fornecida, o interpretador checa se o tipo da definição é igual ao tipo declarado, retornando um erro de tipo caso não seja.

Figura 4.2: Exemplo de programa com a sintaxe do interpretador

```
typevars
  A : *ns ;
  B : *s
end

typedefs
  Bool = 1 + 1 ;

  C = A + B ;

  Cliente  = !Bool.?Bool.End ;
  Servidor = dual(Cliente)
end

vars
  un x : A ;
  un y : C
end

-- Comentário de única linha

{- Comentário de
   múltiplas
   linhas -}

let un false : Bool = inl [1] {} in
let un true  : Bool = inr [1] {} in

let un cliente : Cliente -o Bool =
  \lin c0 : Cliente.
    let lin c1  = send true c0 in
    let {b, c2} = receive c1   in
    let {}      = close c2     in
    b
in

let un servidor : Servidor -o 1 =
  \lin s0 : Servidor.
    let {b, s1} = receive s0 in
    let lin s2  = send b s1  in
    close s2
in

let lin c : Cliente = fork servidor in
cliente c
```

Fonte: O Autor

A sintaxe do interpretador inclui também comentários de única linha, iniciados por `--`, e de múltiplas linhas, delimitados por `{-` e `-}`.

Quando um código é inserido no interpretador, ou carregado a partir de um arquivo, o interpretador verifica se a sintaxe está correta e se o programa é bem-tipado. Caso positivo, o tipo do programa é exibido e então é possível começar a rodar o termo, clicando no botão "Rodar". É importante salientar que, mesmo que o programa seja bem-tipado, sua avaliação pode terminar em um estado de erro, já que o programa pode tentar avaliar uma variável de termo sem definição, inserida na seção `vars`. O programa também pode terminar em um estado de *lock*, quando todas as *threads* em execução estão bloqueadas em uma operação de concorrência.

4.2 Implementação

O código-fonte da implementação está disponível em um repositório *on-line*:

<https://github.com/gabrieldesh/concurrent-lambda-calculus>

O desenvolvimento da aplicação foi realizado na linguagem de programação Elm (<https://elm-lang.org/>). Elm é uma linguagem de programação funcional voltada para o desenvolvimento de aplicações *web* que rodam no navegador. Ela é transpilada para código em JavaScript, o qual pode ser incorporado em uma página HTML. A versão utilizada foi a 0.19.

A aplicação desenvolvida contém cerca de 2000 linhas de código-fonte. Ela é dividida nos seguintes módulos: `AbstractSyntax`, `Parsing`, `TypeInference`, `Evaluation`, `Main` e `Utils`. O módulo `Utils` contém apenas algumas funções auxiliares.

O módulo `AbstractSyntax` define estruturas que representam a árvore de sintaxe abstrata de tipos e termos, com base na sintaxe da Figura 3.1 do capítulo anterior, e a estrutura do programa do interpretador, com as seções de declarações e definições. Também contém funções para exibir *kinds* e tipos como *strings*.

O módulo `Parsing` usa a biblioteca `parser-combinators`, de combinadores de *parser*, para realizar o *parsing* do programa, gerando como resultado a estrutura abstrata de programa definida no módulo `AbstractSyntax`. Os combinadores de *parser* permitem definir *parsers* complexos a partir da composição de outros mais simples.

O módulo `TypeInference` realiza a inferência de tipos do programa, incluindo

a checagem de boa-formação de tipos, substituição de definições de tipos, teste de equivalência de tipos e resolução da operação `dual`.

O módulo `Evaluation` implementa a avaliação *small-step* do programa. A avaliação de termos, correspondente à relação de redução $\longrightarrow_M$ (Figura 3.8 do capítulo anterior) é baseada no algoritmo apresentado por Danvy e Nielsen (2004), para implementação de semânticas operacionais que fazem uso de contextos de avaliação. A avaliação de configurações, correspondente às relações $\equiv$ e $\longrightarrow$ (Figura 3.9 do capítulo anterior), é baseada na máquina abstrata apresentada por Turner (1996, Capítulo 7) para o cálculo-π. O escalonamento de *threads* é feito de forma pseudoaleatória, para que seja possível observar o não-determinismo da linguagem.

Por fim, o módulo `Main` implementa a interface gráfica de usuário, delegando *parsing*, inferência de tipos e avaliação para os demais módulos.

5 CONCLUSÕES E TRABALHO FUTURO

Este trabalho apresentou o projeto de um núcleo de linguagem funcional estendido com operações de concorrência e tipos de sessão. A especificação formal dessa linguagem foi apresentada, definindo de forma precisa os conceitos vistos nos exemplos. Por fim, a especificação serviu de base para a implementação de um interpretador, o qual permite experimentar a linguagem, testar e modificar os exemplos apresentados.

Como foi exemplificado no Capítulo 2, os tipos de sessão permitem especificar protocolos e garantir, via sistema de tipos, que os programas escritos na linguagem seguem esses protocolos. Além disso, os exemplos desse capítulo mostraram como utilizar recursos como recursão geral, tipos recursivos, **new session** e pontos de acesso para suportar de forma bastante abrangente a programação concorrente, permitindo, por exemplo, expressar processos e protocolos repetitivos (seção 2.4), conexões cíclicas (seção 2.5) e condições de corrida (seção 2.6). Porém, como foi também mostrado nesses exemplos, esses recursos introduzem a possibilidade de não-terminação, deadlocks e não-determinismo, comportamentos esses que nem sempre são desejáveis.

O trabalho teve como objetivo tornar o tópico de tipos de sessão mais acessível. Para isso, os exemplos do Capítulo 2 foram construídos com o propósito de serem simples. Contudo, como a linguagem projetada inclui alguns mecanismos relativamente complexos, como tipos recursivos e linearidade, isso tornou a apresentação de alguns exemplos um pouco mais complicada, sendo necessário, por exemplo, o uso de **fold** e **unfold**, e de funções auxiliares para duplicar ou descartar variáveis lineares. Nesse sentido, como trabalho futuro, existem alguns aperfeiçoamentos que poderiam tornar a programação na linguagem mais conveniente, e a apresentação de exemplos mais simples.

A teoria de tipos apresentada neste trabalho é *iso-recursiva*, o que significa que um tipo recursivo e seu *unfolding* não são equivalentes, sendo necessário o uso das operações de **fold** e **unfold** para converter uma expressão de um tipo para o outro. Uma possível melhoria seria eliminar a necessidade dessas operações adotando uma teoria *equi-recursiva*, onde o algoritmo de inferência de tipos consegue reconhecer a equivalência entre os dois tipos. Pierce (2002, Capítulo 21) apresenta de forma bastante detalhada e acessível as questões algoritmicas relacionadas a uma teoria equi-recursiva, fornecendo ideias de como esse recurso poderia ser implementado na linguagem. Seria necessário estudar como essa teoria interage com a noção de dualidade de tipos de sessão.

Outra possível melhoria seria a introdução de algum mecanismo de inferência de

multiplicidade das variáveis, ou seja, das anotações **lin** e **un**. Bernardy et al. (2017) apresentam uma extensão da linguagem de programação Haskell para suportar linearidade, incluindo inferência implícita e polimorfismo paramétrico de multiplicidade. Essas ideias poderiam servir como inspiração para extensões da linguagem apresentada aqui.

Os tipos de sessão têm sido objeto de muito estudo e começam a surgir em linguagens de programação convencionais, tendo um grande potencial de se tornarem um recurso comum no futuro. Espera-se que este trabalho contribua para tornar esse tópico um pouco mais acessível.

REFERÊNCIAS

ABCD. **Session Types in Programming Languages: A Collection of Implementations [online]**. 2017. Disponível em: <https://groups.inf.ed.ac.uk/abcd/session-implementations.html>. Acessado em: 29 de outubro de 2021.

ABRAMSKY, S. Computational interpretations of linear logic. **Theoretical computer science**, Elsevier, v. 111, n. 1-2, p. 3–57, 1993.

ABRAMSKY, S. Proofs as processes. **Theoretical Computer Science**, Elsevier, v. 135, n. 1, p. 5–9, 1994.

ABRAMSKY, S.; GAY, S. J.; NAGARAJAN, R. Interaction categories and the foundations of typed concurrent programming. In: **NATO ASI DPD**. [S.l.: s.n.], 1996. p. 35–113.

BELLIN, G.; SCOTT, P. On the pi-calculus and linear logic. **Theoretical Computer Science**, v. 135, n. 1, p. 11–65, 1994. ISSN 0304-3975. Disponível em: <https://www.sciencedirect.com/science/article/pii/0304397594001049>.

BERNARDY, J.-P. et al. Linear haskell: practical linearity in a higher-order polymorphic language. **Proceedings of the ACM on Programming Languages**, ACM New York, NY, USA, v. 2, n. POPL, p. 1–29, 2017.

BONELLI, E.; COMPAGNONI, A.; GUNTER, E. Correspondence assertions for process synchronization in concurrent communications. **Journal of Functional Programming**, Cambridge University Press, v. 15, n. 2, p. 219–247, 2005.

CAIRES, L.; PFENNING, F. Session types as intuitionistic linear propositions. In: SPRINGER. **International Conference on Concurrency Theory**. [S.l.], 2010. p. 222–236.

CAPECCHI, S. et al. Amalgamating sessions and methods in object-oriented languages with generics. **Theoretical Computer Science**, Elsevier, v. 410, n. 2-3, p. 142–167, 2009.

CARBONE, M. et al. Coherence generalises duality: A logical explanation of multiparty session types. In: SCHLOSS DAGSTUHL-LEIBNIZ-ZENTRUM FUER INFORMATIK. **27th International Conference on Concurrency Theory (CONCUR 2016)**. [S.l.], 2016.

COPPO, M.; DEZANI-CIANCAGLINI, M.; YOSHIDA, N. Asynchronous session types and progress for object oriented languages. In: SPRINGER. **International Conference on Formal Methods for Open Object-Based Distributed Systems**. [S.l.], 2007. p. 1–31.

DANVY, O.; NIELSEN, L. R. Refocusing in reduction semantics. **BRICS Report Series**, v. 11, n. 26, 2004.

DARDHA, O.; GAY, S. J. A new linear logic for deadlock-free session-typed processes. In: SPRINGER. **International Conference on Foundations of Software Science and Computation Structures**. [S.l.], 2018. p. 91–109.

DEZANI-CIANCAGLINI, M. et al. Session types for object-oriented languages. In: SPRINGER. **European Conference on Object-Oriented Programming**. [S.l.], 2006. p. 328–352.

DEZANI-CIANCAGLINI, M. et al. A distributed object-oriented language with session types. In: SPRINGER. **International Symposium on Trustworthy Global Computing**. [S.l.], 2005. p. 299–318.

FÄHNDRICH, M. et al. Language support for fast and reliable message-based communication in singularity os. In: **Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006**. [S.l.: s.n.], 2006. p. 177–190.

GAY, S.; VASCONCELOS, V.; RAVARA, A. **Session types for inter-process communication**. [S.l.], 2003.

GAY, S. J.; VASCONCELOS, V. T. Linear type theory for asynchronous session types. **Journal of Functional Programming**, Cambridge University Press, v. 20, n. 1, p. 19–50, 2010.

GIRARD, J.-Y. Linear logic. **Theoretical computer science**, Elsevier, v. 50, n. 1, p. 1–101, 1987.

HONDA, K. Types for dyadic interaction. In: SPRINGER. **International Conference on Concurrency Theory**. [S.l.], 1993. p. 509–523.

HONDA, K. et al. Scribbling interactions with a formal foundation. In: SPRINGER. **International Conference on Distributed Computing and Internet Technology**. [S.l.], 2011. p. 55–75.

HONDA, K.; VASCONCELOS, V. T.; KUBO, M. Language primitives and type discipline for structured communication-based programming. In: SPRINGER. **European Symposium on Programming**. [S.l.], 1998. p. 122–138.

HONDA, K.; YOSHIDA, N.; CARBONE, M. Multiparty asynchronous session types. In: **Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages**. [S.l.: s.n.], 2008. p. 273–284.

LINDLEY, S.; MORRIS, J. G. A semantics for propositions as sessions. In: SPRINGER. **European Symposium on Programming Languages and Systems**. [S.l.], 2015. p. 560–584.

LINDLEY, S.; MORRIS, J. G. Lightweight functional session types. **Behavioural Types: from Theory to Tools. River Publishers**, p. 265–286, 2017.

MILNER, R. **Communication and Concurrency**. [S.l.]: Prentice Hall, 1989.

MILNER, R.; PARROW, J.; WALKER, D. A calculus of mobile processes, partes i e ii. **Information and computation**, Elsevier, v. 100, n. 1, p. 1–77, 1992.

PIERCE, B. C. **Types and programming languages**. [S.l.]: MIT press, 2002.

PRAUN, C. von. Race conditions. In: PADUA, D. (Ed.). **Encyclopedia of Parallel Computing**. Boston, MA: Springer US, 2011. p. 1691–1697. ISBN 978-0-387-09766-4. Disponível em: <https://doi.org/10.1007/978-0-387-09766-4_36>.

Scribble Team. **Página inicial do projeto Scribble [online]**. 2015. Disponível em: <http://www.scribble.org>. Acessado em: 5 de novembro de 2021.

TAKEUCHI, K.; HONDA, K.; KUBO, M. An interaction-based language and its typing system. In: SPRINGER. **International Conference on Parallel Architectures and Languages Europe**. [S.l.], 1994. p. 398–413.

TURNER, D. **The polymorphic pi-calculus: Theory and implementation**. Tese (Doutorado) — University of Edinburgh. College of Science and Engineering. School of Informatics, 1996.

TURNER, D. N.; WADLER, P. Operational interpretations of linear logic. **Theoretical Computer Science**, Elsevier, v. 227, n. 1-2, p. 231–248, 1999.

VALLECILLO, A.; VASCONCELOS, V. T.; RAVARA, A. Typing the behavior of software components using session types. **Fundamenta Informaticæ**, IOS Press, v. 73, n. 4, p. 583–598, 2006.

VASCONCELOS, V.; RAVARA, A.; GAY, S. Session types for functional multithreading. In: SPRINGER. **International Conference on Concurrency Theory**. [S.l.], 2004. p. 497–511.

VASCONCELOS, V. T.; GAY, S. J.; RAVARA, A. Type checking a multithreaded functional language with session types. **Theoretical Computer Science**, Elsevier, v. 368, n. 1-2, p. 64–87, 2006.

W3C. **Web Services Choreography Description Language Version 1.0 [online]**. 2005. Disponível em: <https://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/>. Acessado em: 29 de outubro de 2021.

WADLER, P. A taste of linear logic. In: SPRINGER. **International Symposium on Mathematical Foundations of Computer Science**. [S.l.], 1993. p. 185–210.

WADLER, P. Propositions as sessions. **ACM SIGPLAN Notices**, ACM New York, NY, USA, v. 47, n. 9, p. 273–286, 2012.

YOSHIDA, N.; VASCONCELOS, V. T. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. **Electronic Notes in Theoretical Computer Science**, Elsevier, v. 171, n. 4, p. 73–93, 2007.