ESPECIFICAÇÃO DE SISTEMAS UTILIZANDO LÓGICA LINEAR COM SUBEXPONENCIAIS

GISELLE MACHADO N. REIS

ESPECIFICAÇÃO DE SISTEMAS UTILIZANDO LÓGICA LINEAR COM SUBEXPONENCIAIS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Elaine Gouvêa Pimentel
Co-Orientador: Roberto da Silva Bigonha

Belo Horizonte - MG

Setembro de 2010

Reis, Giselle Machado N.

R375e Especificação de sistemas utilizando lógica linear com subexponenciais / Giselle Machado N. Reis. — Belo Horizonte - MG, 2010
xx, 68 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais. Departamento de Ciência da Computação.

Orientadora: Elaine Gouvêa Pimentel
Co-Orientador: Roberto da Silva Bigonha

1. Linguagem e lógica - Tese. 2. Programação lógica - Tese. I. Orientador II. Título.

CDU 519.6*53 (043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Especificação de sistemas utilizando lógica linear com subexponencias

GISELLE MACHADO NOGUEIRA REIS

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROFA. ELAINE GOUVÊA PIMENTEL - Orientadora
Departamento de Matemática - UFMG

PROF. ROBERTO DA SILVA BIGONHA - Co-orientador
Departamento de Ciência da Computação - UFMG

PROF. EDWARD HERMANN HAEUSLER
Departamento de Informática - PUC-RJ

PROF. FERNANDO MAGNO QUINTÃO PEREIRA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 17 de novembro de 2010.

Agradecimentos

Este trabalho foi fruto de dois anos (ou mais) de estudos constantes. Ele não teria sido possível sem a participação e empenho de algumas pessoas fundamentais.

Muito obrigada Elaine, pela orientação, pelas aulas e pelos conselhos. Sem você eu não teria chegado até aqui.

Muito obrigada Monteiro, por ter me apresentado à Elaine!

Muito obrigada Vivek, por todo o trabalho desenvolvido anteriormente, pela excelente tese de doutorado que foi o início de tudo e por todos os esclarecimentos de dúvidas quando precisei.

Muito obrigada Bigonha, por me acolher no laboratório de Linguagens de Programação, pelas aulas e pelas monitorias, onde eu aprendi bastante sobre o que é ser professor.

Muito obrigada mamãe e papai por fazer tudo que esteve ao alcance de vocês por mim. Vocês sabem que vai ser sempre mais do que eu mereço.

Muito obrigada João, pelo companheirismo, pelo carinho, pelo amor e pelas risadas. Os últimos dois anos teriam sidos muito chatos sem você ao meu lado.

Obrigada a todos os professores do DCC que, direta ou indiretamente, contribuíram para minha formação, profissional e pessoal.

Obrigada aos funcionários do DCC, sem os quais nós alunos, e também os professores, ficariam perdidos.

"A resposta certa, não importa nada: o essencial é que as perguntas estejam certas."
(Mário Quintana)

Resumo

A programação lógica é caracterizada principalmente pela interpretação de fórmulas lógicas como programas. Essas fórmulas são utilizadas para provar objetivos e essas provas são interpretadas como a execução do programa (computação). Esse paradigma é interessante pela formalidade das especificações, herdada da lógica utilizada, o que facilita a prova de algumas propriedades importantes que não estariam tão claras se o programa fosse implementado imperativamente, por exemplo.

Entretanto, uma lógica precisa de algumas características essenciais para ser a base de uma linguagem de programação. Mais especificamente, é necessário que a busca por provas possa ser feita de maneira "bem-comportada", no sentido de existir uma forma determinística de fazer essa busca. Devido a essa restrição, somente algumas lógicas, ou mesmo fragmentos de lógicas, podem ser implementados como linguagens de programação. Como consequência disso essas linguagens ficam também limitadas, muitas vezes não apresentando aspectos básicos como modularização, abstração de dados ou concorrência. A linguagem de programação lógica mais popular, Prolog, é um exemplo claro da existência de tais problemas.

Desde a popularização desse paradigma de programação, foram feitas várias tentativas no sentido de aumentar a expressividade da lógica utilizada para se obter uma linguagem mais completa e ainda puramente lógica. Entre as contribuições mais significativas estão LO [Andreoli & Pareschi, 1991] e λ-Prolog [Nadathur & Miller, 1988], cujas lógicas conseguem capturar a modularização, abstração de dados e alguns aspectos de concorrência. Porém, as lógicas que implementam essas linguagens ainda são fragmentos (da lógica linear, no caso de LO, e da lógica clássica, no caso de λ-Prolog).

A impossibilidade de se implementar uma linguagem de programação com toda a lógica clássica motivou o estudo da lógica linear, que pode ser completamente utilizada para se implementar uma linguagem. Recentemente foi proposto um refinamento da lógica linear, caracterizado pelo uso de subexponenciais, que podem ser interpretados como *locations* (repositórios de dados). Com essa nova lógica, foi possível provar a correspondência entre algoritmos e busca por provas.

Essa dissertação de mestrado apresenta uma implementação para a lógica linear com subexponenciais. Além disso, para mostrar o aumento de expressividade da lógica, estão codificados alguns sistemas de provas na linguagem implementada. Também é mostrada a implementação de uma linguagem imperativa cuja especificação semântica dos comandos é feita utilizando a lógica linear com subexponenciais. Dessa forma, é possível observar de maneira clara como os algoritmos correspondem à busca por provas.

Abstract

Logic programming is defined as the use of logic formulas representing programs and proof search of these formulas as the execution of the program (computation). This is an interesting paradigm because of the specifications' formality, which is inherited from the logic itself and facilitates the proof of some properties that would not be so obvious if the program was written in an imperative programming language, for example.

However, a logic needs to have some important characteristics to be the basis of a programming language. Namely, it is necessary that the proof search is "well-behaved", meaning that there is a deterministic procedure to look for proofs. Due to this restriction, only a few logics, or parts of it, can be implemented as a programming language. As a consequence, these languages are also limited, not having some basic features such as modularization, data abstraction or concurrency. The most popular logic programming language, Prolog, is an example of such limited programming languages.

Since the popularization of the logic programming paradigm, numerous attempts have been made to increase the expressiveness of the underlying logic, so that the programming language obtained has more features yet still purely logic. Among the most significant contributions are LO [Andreoli & Pareschi, 1991] and λ-Prolog [Nadathur & Miller, 1988], whose logics can capture modularization, data abstraction and some aspects of concurrency. Nevertheless, the logics implemented are still fragments (of linear and classical logic, respectively).

The impossibility to implement a programming language with the whole classical logic is the main motivation for the study of linear logic, which, in contrast, is a programming language itself. Recently a refinement of the linear logic was proposed with the use of subexponentials, interpreted as *locations* in the language. With this new logic, the correspondence between algorithms and proof search could be proved.

This master dissertation presents an implementation for the linear logic with subexponentials. In order to show the increase of expressiveness of the logic, it is shown the codification of some sequent calculus proof systems in the new language. It also presents the implementation of a simple imperative programming language. The semantics of this language is described using linear logic with subexponentials, therefore it is possible to execute a program using this logic's implementation and it becomes clear the correspondence between algorithms and proof search.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

O papel principal da computação é a resolução de problemas de maneira automatizada. Para isso, existem algoritmos (sequências finitas de passos) que processam dados de entrada e apresentam uma saída ao finalizar esse processamento. Os dados de entrada são, em geral, uma instância do problema que se quer resolver. Considere, por exemplo, o problema do caixeiro viajante: percorrer um conjunto de cidades de modo que a distância total viajada seja a menor possível. Os dados de entrada para esse problema (informações necessárias para a resolução) seriam o conjunto de cidades, as distâncias entre as mesmas e aquelas que devem ser percorridas. Isso caracteriza uma instância do problema. Um algoritmo pode ser implementado de modo que ele resolva qualquer instância, bastando apenas alterar os dados de entrada.

Entretanto, para que a máquina consiga resolver esse problema é necessário que o algoritmo (passos para resolução) e os dados de entrada sejam representados de modo apropriado. A linguagem natural, embora funcione muito bem para humanos, é muito complexa e aberta a interpretações para uma máquina. Dessa maneira, é preciso "traduzir" o problema para uma linguagem mais formal e exata, que possa ser processada por uma máquina. Essa tradução é feita em vários níveis até chegar no conjunto de instruções do processador da máquina, que, por sua vez, é executado com alterações de alta e baixa voltagens no sinal elétrico (1's e 0's). A primeira parte da tradução consiste na abstração dos dados do problema em entidades matemáticas (funções, relações, restrições, grafos, etc.). Em geral, isso facilita a escolha do método de resolução, já que existem algoritmos comprovadamente eficientes para resolução de problemas mais gerais nessas estruturas. No caso do nosso exemplo, a abstração feita é a representação das cidades como vértices em um grafo e da distância entre elas como arestas com peso entre as cidades. A partir desse momento, podemos escolher um dos vários algoritmos em grafos existentes para calcular a menor distância entre um conjunto de vértices (ou cidades). Essa etapa é em geral realizada manualmente, sem a ajuda de um computador. Ela exige uma certa experiência e conhecimento de algoritmos.

A próxima etapa consiste na tradução dessa abstração e algoritmo para uma linguagem que o computador seja capaz de processar. Essa linguagem, na verdade, será um sistema formal. Um sistema formal é composto por uma linguagem formal e um conjunto de regras de reescrita, que basicamente indicam qual o processamento

que deve ser realizado para cada construto da linguagem. Existem vários tipos desses sistemas que podem ser utilizados na implementação de um algoritmo. As linguagens de programação são um exemplo. A linguagem formal, nesse caso, é a sintaxe e as regras de reescrita são definidas no compilador ou interpretador da linguagem. Outro exemplo de sistema formal é o λ-calculus. Novamente a linguagem formal é a sintaxe, representada pela gramática:

$$T := \langle const \rangle | \langle var \rangle | (TT) | \lambda \langle var \rangle . T$$

As regras de reescrita para o λ-calculus são: α-conversão, β-redução e η-redução. Nessa dissertação iremos tratar de um outro sistema formal: lógica. A linguagem formal de uma lógica é a sua sintaxe, em geral formada por átomos, fórmulas e conectivos lógicos. As regras de reescrita são as regras de inferência de cada conectivo lógico, que indicam o que se pode concluir dado um conjunto de premissas (e.g. se sabemos que ambos os átomos A e B são verdadeiros, pode-se concluir corretamente que $A \wedge B$ é verdadeiro).

Após a tradução da abstração em um sistema formal, a tradução desses comandos para linguagem de máquina está geralmente automatizada por um compilador ou interpretador.

1.1 Lógica como sistema formal

A lógica pode ser utilizada basicamente de duas maneiras na computação [Miller, 1999]. A primeira opção seria utilizá-la para justificar transições de estado. Cada transição é associada a uma expressão lógica e o fluxo de execução realiza essa transição somente se a expressão for verdadeira. Esse paradigma é chamado de "computação como modelo".

A segunda maneira é representar os próprios estados como proposições lógicas, e a mudança de estado é modelada pela aplicação de uma regra de reescrita que irá transformar essa proposição em outra (ou outras). Nesse modelo a computação (execução de um programa) é representada pelo desenvolvimento de uma prova da proposição inicial, que representa o estado inicial do sistema (algoritmo, variáveis, dados de entrada, etc.). Esse paradigma é chamado de "computação como dedução".

Nos últimos anos, têm sido concentrados esforços para aproximar as lógicas de uma linguagem de programação, de modo que se possa utilizar a computação como dedução. A vantagem em se utilizar esse paradigma é o fato das computações e algoritmos herdarem o formalismo de uma lógica, o que permite que sejam provadas afirmações sobre essas especificações de maneira mais automática, utilizando tautologias já conhecidas.

No modelo de computação como dedução, uma prova corresponde à execução de um algoritmo, e por isso, não é qualquer prova que pode ser utilizada como computação. A seguir é apresentado o conceito de prova construtiva, e algumas condições para que um sistema de provas de uma lógica possa ser implementado como uma linguagem de programação.

Uma prova construtiva é uma sequência de passos (algoritmo) que irá mostrar como uma proposição pode ser construída. Se isso for possível, prova-se que ela é

verdadeira, já que existe um modo de obtê-la. Em provas construtivas o princípio do terceiro excluído ($a \vee \neg a$) não pode ser utilizado. Dizer que uma proposição é verdadeira simplesmente porque ela não é falsa não apresenta uma maneira de construí-la. Do mesmo modo, prova por contradição também não é uma prova construtiva.

As provas correspondentes a computações devem ser construtivas. Dado um estado inicial S_0, queremos provar que outro estado S_n é atingido. Isso equivale a provar a proposição S_n dado que S_0 é verdadeiro, i.e., S_0 é o contexto da prova, conjunto de proposições a partir das quais S_n será derivado. A execução de um algoritmo é uma sequência de passos que leva o programa do estado S_0 para o estado S_n. Para que uma prova modele essa execução, ela também deve apresentar uma sequência de passos que chegue em S_n.

1.2 Cálculo de sequentes

O cálculo de sequentes, proposto por Gentzen em 1935 [Gentzen, 1969], é um formalismo utilizado para expressar e estudar a estrutura de provas de sistemas lógicos. Um sistema de cálculo de sequentes é composto por um conjunto de *regras de inferência* que derivam *sequentes*. Um sequente é uma estrutura da seguinte forma:

$$B_1, B_2, ..., B_n \vdash C_1, C_2, ..., C_k$$

Em que B_i e C_j são fórmulas na lógica considerada e $n, k \geq 0$ (ou seja, um dos ou ambos os lados podem ser vazios). Ele é interpretado como: "Dadas as fórmulas $B_1, B_2, ..., B_n$, podemos concluir $C_1, C_2, ..., C_k$ na lógica considerada".

Uma regra de inferência do cálculo de sequentes é escrita com um sequente, uma linha horizontal acima e zero ou mais sequentes acima da linha horizontal. Chamamos os sequentes que ocorrem acima da linha horizontal de premissas e o sequente abaixo da linha horizontal é a conclusão. Os axiomas, em particular, não possuem premissas; outras regras porém, podem possuir uma ou mais premissas. A Figura 1.1 mostra a regra de inferência para o conectivo $\wedge_r$ na lógica clássica. Ela pode ser interpretada da seguinte maneira: dado que de um conjunto de fórmulas Γ é derivado P e outro conjunto Δ, e do mesmo conjunto Γ é derivado Q e o mesmo conjunto Δ, pode-se afirmar que de Γ se deriva $P \wedge Q$ e, obviamente, Δ.

$$\frac{\Gamma \vdash P, \Delta \quad \Gamma \vdash Q, \Delta}{\Gamma \vdash P \wedge Q, \Delta} \wedge_r$$

Figura 1.1. Regra de inferência para o conectivo $\wedge_r$ na lógica clássica.

As regras de inferência podem ser divididas em três tipos:

- Regras de identidade: aquelas que precisam verificar se duas fórmulas são iguais (e.g., axioma inicial e regra *cut*).
- Regras estruturais: aquelas que alteram a estrutura de um sequente, sem operar nos conectivos lógicos (e.g., *weakening* e *contraction*).

- Regras lógicas: aquelas que decompõem os conectivos lógicos.

Uma derivação no cálculo de sequente é uma árvore, resultado da aplicação sucessiva de regras de inferência nos sequentes. Uma prova é uma derivação que inicia-se do sequente que se quer provar e na qual todos os ramos da árvore terminam com o axioma inicial. Essa prova é construída de baixo para cima, e uma regra de inferência sempre pode ser aplicada a um sequente quando o mesmo é a conclusão dessa regra. Observe que é possível que mais de uma regra possa ser aplicada em um sequente. Cada aplicação atua em uma fórmula de cada vez, seja para decompô-la, duplicá-la ou removê-la, essa é a *fórmula principal* do sequente.

Se a negação é involutiva ($\neg\neg a \equiv a$) na lógica considerada, prefere-se utilizar o cálculo de sequentes *one-sided* devido à sua simplicidade. Nessa versão do cálculo, todas as fórmulas ocorrem do lado direito do símbolo $\vdash$, sendo que aquelas que estavam do lado esquerdo estão na sua forma negada. O cálculo de sequentes para a lógica linear com subexponenciais utilizado nesse trabalho é *one-sided*.

1.3 Implementação de interpretadores lógicos

A especificação de sistemas usando lógica é justificada pelo formalismo herdado da lógica, que facilita a prova de propriedades utilizando as regras de inferência independente do escopo do problema. A existência de provadores de teorema automáticos ou semi-automáticos é essencial para realizar essas tarefas. Esses provadores são implementados como interpretadores para lógicas específicas. Prolog, por exemplo, é um interpretador lógico para um fragmento da lógica intuicionista (dado pelas cláusulas de Horn - Seção 3.2.1). λ-Prolog é um interpretador para um fragmento maior da mesma lógica (dado pelas fórmulas de Harrop - Seção 3.2.2). Lolli [Hodas & Miller, 1994], por outro lado, é um interpretador para um subconjunto da lógica linear, mas especificamente, a lógica linear intuicionista. Existem implementados diversos interpretadores para as mais variadas lógicas.

Entretanto, a implementação de um interpretador lógico apresenta alguns problemas não triviais. Dado um sistema de provas de uma lógica, a sua tradução direta para uma linguagem de programação não garante de modo algum que ela será capaz de provar qualquer coisa provável pelo sistema. As principais dificuldades estão relacionadas com o não-determinismo da aplicação de regras ou escolha de fórmulas. Quando uma prova é feita "à mão", uma pessoa é capaz de analisar todo o contexto e as opções para continuação e fazer a escolha correta. A pessoa pode enxergar com certa facilidade qual regra deve ser aplicada. Por outro lado, uma máquina não é capaz de decidir tal coisa. Devido a essas dificuldades, muitos esforços foram despendidos para desenvolver uma estratégia de prova que seja correta e completa. Isso significa que, utilizando tal estratégia, é possível encontrar uma prova para uma fórmula se ela existe, e toda prova encontrada no sistema é uma prova válida da lógica. Nessa seção, alguns dos problemas da implementação de interpretadores lógicos são explicados.

Durante a prova de um sequente, em certo ponto, é necessário escolher uma regra de inferência para ser aplicada dentre várias possíveis. Essa escolha pode levar a um sucesso ou falha da prova. Se a falha pode ser detectada, i.e., atinge-se um ponto tal

que nenhuma regra de inferência é aplicável, é realizado o *backchaining* até o último ponto em que uma escolha não-determinística foi feita e escolhe-se outro caminho para continuar. O problema são as falhas que não são facilmente detectadas, tais como o *loop* infinito.

Esses *loops* estão relacionados, na maior parte das vezes, com a sequência de fórmulas e regras de inferência escolhidas. Observe, por exemplo, a regra *contraction*. Aplicando essa regra, é sempre possível duplicar uma fórmula do sequente. Uma busca cega por uma prova pode seguir aplicando essa regra de inferência indefinidamente, como é mostrado abaixo, e nunca terminar.

$$\frac{\Gamma \vdash \Delta, P, P}{\Gamma \vdash \Delta, P}\, C_r \qquad \frac{\dfrac{\dfrac{\vdots}{P \vee Q \vdash P, Q, Q, Q}\, C_r}{P \vee Q \vdash P, Q, Q}\, C_r}{P \vee Q \vdash P, Q}\, C_r$$

A estratégia utilizada para escolher qual a próxima regra a ser aplicada e qual a fórmula principal pode então determinar se a prova irá entrar em *loop* ou não, logo essa estratégia deve ser escolhida cuidadosamente. Ela deve minimizar ao máximo o não-determinismo de modo a evitar *loops*.

Para garantir a correção e completude de um sistema de provas, deve ser definida uma **estratégia uniforme de prova**, que irá guiar a escolha das regras de inferência e fórmulas principais e fazer com que a busca por prova seja mais determinística. Outra vantagem em se ter uma disciplina para a busca de provas é a redução da redundância, diminuindo o espaço de busca e tornando o processo mais eficiente. Duas provas são ditas redundantes, ou equivalentes, quando uma pode ser obtida da outra a partir de uma simples reordenação de aplicação de regras de inferência.

Andreoli desenvolveu uma estratégia de prova correta e completa para a lógica linear, chamada *focusing*, que se baseia na classificação dos conectivos dessa lógica em *assíncronos* e *síncronos* [Andreoli, 1992]. Essa estratégia é explicada com detalhes na Seção 4.3.

Uma outra causa importante do não-determinismo durante a busca por provas é a utilização da regra *cut*, mostrada na Figura 1.2.

$$\frac{\Gamma_1 \vdash P, \Delta_1 \qquad \Gamma_2, P \vdash \Delta_2}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ [\text{Cut}]$$

Figura 1.2. Regra *cut* para a lógica clássica *LK*.

A aplicação dessa regra de inferência envolve a introdução de uma nova fórmula P, que não estava presente na conclusão. Existem infinitas fórmulas na lógica para se escolher, e é portanto impossível que uma máquina tente todas as possibilidades exaustivamente. Felizmente Gentzen mostrou que qualquer prova que utilize uma instância da regra *cut* pode ser transformada em uma outra prova que não possui a aplicação da mesma, chamada de *cut-free* [Gentzen, 1969], na lógica clássica. Em uma lógica com essa propriedade, a regra *cut* é dita ser admissível. Girard provou que o *cut* é

admissível na lógica linear [Girard, 1987], e esse é o primeiro passo para que a lógica possa ser uma base adequada para uma linguagem de programação.

Capítulo 2

Revisão da literatura

2.1 Principais trabalhos

Para que uma lógica seja a base de uma linguagem de programação lógica, é necessário que a busca por provas nessa lógica, feita de uma forma determinística, seja completa. Isso significa que deve ser possível encontrar todas as provas na lógica utilizando tal estratégia. Infelizmente, as lógicas clássica e intuicionista não apresentam uma estratégia de provas determinística completa. Por isso, as linguagens de programação que utilizam tais lógicas fazem uso de apenas um fragmento das mesmas, como será mostrado nas Seções 3.2.1 e 3.2.2. Esse fato limita a expressividade da linguagem, já que nem todos os sequentes possíveis da lógica podem ser representados ou provados na implementação da linguagem.

Considerando que um programa lógico é um conjunto de fórmulas Γ, e sua entrada (o que se quer provar) seja outro conjunto Δ, a execução é correspondente à prova do seguinte sequente: $\Gamma \vdash \Delta$. A expressividade de uma linguagem de programação lógica pode ser medida pela quantidade de transformações que podem sofrer cada um desses conjuntos. Quanto mais for possível manipulá-los, mais expressiva será a linguagem.

Prolog é a implementação das cláusulas de Horn, um subconjunto da lógica clássica cujos objetivos (*goals* - conjunto Δ) são formados apenas por átomos ou conjunções de átomos. Um dos principais problemas das cláusulas de Horn é que todos os objetivos devem ser derivados do mesmo programa (conjunto Γ), e todas as cláusulas necessárias para provar um objetivo já devem estar presentes no programa principal. Logo, o sequente de uma execução em Prolog tem o conjunto Γ estático, e Δ está reduzido a apenas uma fórmula atômica. Utilizando uma interpretação de linguagens de programação, a imutabilidade de Γ é equivalente a dizer que o programa seria composto de um módulo somente, ou seja, não existe modularização.

A linguagem de programação λ-Prolog é uma extensão do Prolog e implementa também um fragmento da lógica clássica: as fórmulas de Harrop. A principal alteração (dentre outras importantes, como a utilização de tipos) é a permissão de implicações nos objetivos (Δ), o que dá alguma idéia de modularização. Assim, provar $A \supset B$ com contexto Γ é equivalente a provar B com contexto $\Gamma \cup \{A\}$. Fica claro dessa maneira que o que está contido no "módulo" A é utilizado somente na prova de B.

O programa (Γ) pode então ter fórmulas acrescentadas a ele em tempo de execução, mas o objetivo Δ ainda é formado por apenas uma fórmula, apesar de a implicação ser permitida. Porém, a adição de módulos ao contexto à medida que ocorrem implicações no objetivo faz com que ele seja aumentado indefinidamente.

Foi provado que as fórmulas de Harrop são o conjunto maximal da lógica clássica que possui uma estratégia uniforme de prova, característica essencial para que a lógica seja a base de uma linguagem de programação. Portanto, para se ter uma linguagem de programação lógica mais robusta devemos mudar de lógica. A lógica linear foi utilizada com esse objetivo [Miller, 1994]. Ela é dita ser uma lógica de recursos finitos, pois as proposições podem ser consumidas durante uma prova, fazendo assim com que o contexto não seja mais estritamente crescente como no caso das fórmulas de Harrop da lógica clássica. Dessa forma, o conjunto Γ pode aumentar ou diminuir à medida que o programa é executado. O consumo de proposições significa que elas não podem ser utilizadas quantas vezes se quer para derivação (lógica de recursos conscientes).

Uma das primeiras implementações utilizando a lógica linear foi a linguagem Lolli [Hodas & Miller, 1994]. Ela implementa um fragmento intuicionista da lógica linear, sem os conectivos $\parr$ (disjunção multiplicativa) e ! (bang[1]). Após algumas alterações na formulação devido a essas restrições, tem-se um sistema de provas uniforme e é possível, portanto, implementar uma linguagem lógica de programação.

Todas as linguagens citadas até agora lidam com apenas uma conclusão (uma única fórmula no conjunto Δ). Uma direção natural seria trabalhar com múltiplas conclusões, que é o que Miller propõe com a linguagem Forum [Miller, 1994]. Essa linguagem pode ser considerada uma fusão de Lolli com LO [Andreoli & Pareschi, 1991], incorporando a abstração de dados da primeira com os princípios de concorrência da segunda. Um dos problemas encontrados foi a possibilidade da aplicação de mais de uma regra no lado direito a cada passo da prova (já que existem múltiplas conclusões). Isso foi resolvido com a utilização de um sistema de provas focado, proposto por Andreoli [Andreoli, 1992]. Nesse sistema, a prova é dividida em duas fases (assíncrona e síncrona) e tem-se uma estratégia de prova uniforme, permitindo assim que a lógica seja base para uma linguagem de programação.

A lógica linear apresenta algum controle sobre o contexto, resolvendo o problema do seu crescimento constante nas fórmulas de Harrop. Isso é feito com a utilização de subexponenciais: para que o operador ! seja retirado de uma fórmula com a qual se está trabalhando no momento (focada) é necessário que o contexto linear seja vazio. Entretanto esse contexto é único, e ele deve estar completamente vazio. Muitas vezes temos problemas que são formados por entidades distintas e desejamos que o conjunto de apenas algumas delas seja vazio, e não todas. Por exemplo, em um problema utilizando grafos muitas vezes é necessário verificar se o conjunto de vértices ou arestas está vazio, mas não ambos. Para se ter um maior controle, Nigam [Nigam, 2009] propôs a utilização de subexponenciais, que consiste basicamente na divisão do operador ! em vários ($!^1, !^2, !^3, ...$) com uma ordenação parcial entre eles. Isso permite que existam restrições do tipo: para retirar um operador $!^i$, não devem existir fórmulas com operador

[1] Colocado ao lado esquerdo de uma implicação, esse operador faz com que a interpretação da fórmula seja clássica, e assim podemos usar o lado esquerdo quantas vezes for preciso para derivar o lado direito.

$!^j$ tal que j seja menor (ou não relacionado, já que a ordenação é parcial) que i. Isso é o mesmo que dizer que um subconjunto do contexto (aquele cujas fórmulas têm o modificador $!^j$) seja vazio. Nigam apresenta a lógica linear com os novos operadores e um sistema de prova para a mesma.

2.2 Soluções existentes

Atualmente não existe ainda uma implementação para a lógica linear com subexponenciais devido ao fato da mesma ter sido proposta em uma tese de doutorado defendida em setembro de 2009. Existem entretanto outras implementações da lógica linear (sem subexponenciais), como por exemplo: Lolli (fragmento intuicionista da lógica linear) [Hodas & Miller, 1994], LO [Andreoli & Pareschi, 1991] e Forum [Miller, 1994].

Capítulo 3

Lógicas

3.1 Lógica Clássica

A lógica clássica lida com verdades estáveis, onde uma afirmativa ou é verdadeira ou falsa, em que falso e verdadeiro são duais. A sintaxe da lógica clássica pode ser descrita pela seguinte gramática, em que A é um átomo e F é uma fórmula:

$$F ::= A \mid \neg F \mid F \wedge F \mid F \vee F \mid F \Rightarrow F \mid \bot \mid \forall x.F \mid \exists x.F$$

O sistema de cálculo de sequentes para a lógica clássica é apresentado na Figura 3.1.

O princípio do terceiro excluído é válido na lógica clássica, e sua prova é exibida na Figura 3.2. Por esse motivo, provas por contradição são válidas nessa lógica. Ela é muito utilizada na matemática, em que cada afirmação ou possui uma prova ou um contra-exemplo que a torna falsa. Observe por exemplo a seguinte afirmação:

Teorema 1. *Existem dois números irracionais x e y tais que x^y é racional.*

Prova 1. *Sabemos que $\sqrt{2}$ é irracional e que 2 é racional. Considere o número $\sqrt{2}^{\sqrt{2}}$. Ele certamente é racional ou irracional. Se ele for racional, podemos escolher $x = y = \sqrt{2}$, e o teorema vale. Se ele for irracional, podemos escolher $x = \sqrt{2}^{\sqrt{2}}$ e $y = \sqrt{2}$ e temos:*
$x^y = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{(\sqrt{2}\times\sqrt{2})} = \sqrt{2}^2 = 2$
Como 2 é racional, o teorema também vale.

Por meio da prova do teorema não é possível saber se $\sqrt{2}^{\sqrt{2}}$ é racional ou irracional, e não se consegue de fato dois números x e y que satisfaçam a propriedade. Mesmo assim, essa é uma prova válida, e não há dúvidas de que o teorema é verdadeiro. Essa é uma prova altamente não construtiva. Provas desse tipo são válidas apenas em sistemas em que falso e verdadeiro são duais, ou seja, qualquer afirmação é sempre falsa ou verdadeira. No caso acima foi utilizado o fato de que um número ou é racional ou é irracional, não havendo outra possibilidade.

REGRAS DE IDENTIDADE

$$\frac{}{P \vdash P}\ [\mathrm{I}] \qquad \frac{\Gamma_1 \vdash P, \Delta_1 \qquad \Gamma_2, P \vdash \Delta_2}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ [\mathrm{Cut}]$$

REGRAS LÓGICAS

$$\frac{}{\bot, \Gamma \vdash \Delta}\ [\bot] \qquad \frac{}{\Gamma \vdash \Delta, \top}\ [\top]$$

$$\frac{\Gamma \vdash \Delta, P}{\Gamma, \neg P \vdash \Delta}\ [\neg_l] \qquad \frac{\Gamma, P \vdash \Delta}{\Gamma \vdash \neg P, \Delta}\ [\neg_r]$$

$$\frac{P_i, \Gamma \vdash \Delta}{P_1 \wedge P_2, \Gamma \vdash \Delta}\ [\wedge_{li}] \qquad \frac{\Gamma \vdash \Delta, P \quad \Gamma \vdash \Delta, Q}{\Gamma \vdash \Delta, P \wedge Q}\ [\wedge_r]$$

$$\frac{P, \Gamma \vdash \Delta \quad Q, \Gamma \vdash \Delta}{P \vee Q, \Gamma \vdash \Delta}\ [\vee_l] \qquad \frac{\Gamma \vdash \Delta, P_i}{\Gamma \vdash \Delta, P_1 \vee P_2}\ [\vee_{ri}]$$

$$\frac{\Gamma_1 \vdash \Delta_1, P \quad Q, \Gamma_2 \vdash \Delta_2}{P \Rightarrow Q, \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ [\Rightarrow_l] \qquad \frac{\Gamma, P \vdash \Delta, Q}{\Gamma \vdash \Delta, P \Rightarrow Q}\ [\Rightarrow_r]$$

$$\frac{P[c/x], \Gamma \vdash \Delta}{\exists x.P, \Gamma \vdash \Delta}\ [\exists_l] \qquad \frac{\Gamma \vdash P[t/x], \Delta}{\Gamma \vdash \exists x.P, \Delta}\ [\exists_r]$$

$$\frac{P[t/x], \Gamma \vdash \Delta}{\forall x.P, \Gamma \vdash \Delta}\ [\forall_l] \qquad \frac{\Gamma \vdash \Delta, P[c/x]}{\Gamma \vdash \Delta, \forall x.P}\ [\forall_r]$$

REGRAS ESTRUTURAIS

$$\frac{P, P, \Gamma \vdash \Delta}{P, \Gamma \vdash \Delta}\ [C_l] \qquad \frac{\Gamma \vdash \Delta, P, P}{\Gamma \vdash \Delta, P}\ [C_r]$$

$$\frac{\Gamma \vdash \Delta}{P, \Gamma \vdash \Delta}\ [W_l] \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, P}\ [W_r]$$

Figura 3.1. Cálculo de sequentes para a lógica clássica. Nas regras $\forall_r$ e $\exists_l$, a variável c não ocorre livre em Γ ou Δ.

Porém, nem todos os sistemas apresentam essa propriedade e a prova pelo princípio do terceiro excluído não pode ser sempre aplicada, como é o caso do seguinte teorema:

Teorema 2. *A soma dos n primeiros números naturais é* $\frac{n(n+1)}{2}$.

A prova por contradição desse teorema envolveria mostrar que a soma não é igual a nenhuma das outras expressões envolvendo uma variável n, o que é uma tarefa impossível. Nesse caso seria melhor utilizar uma prova por indução ou uma prova construtiva mostrando como se chegou à expressão. Em geral, provas por contradição são impossíveis de serem aplicadas quando se tem o conceito de infinito envolvido. No exemplo acima existem infinitas outras equações em n.

O mesmo ocorre quando utilizamos a lógica para modelar computações. Nesse cenário, cada aplicação de regra de inferência corresponde a um passo do algoritmo, e o que se quer provar é que um estado do sistema (objetivo) é alcançável a partir do estado inicial. Como o conjunto de estados possíveis de um sistema é potencialmente infinito, as provas por contradição não são admissíveis, e o que se deseja realmente é que exista uma prova construtiva que demonstre os passos seguidos para que o estado objetivo seja alcançado, assim como um algoritmo faria. O princípio do terceiro excluído, provável na lógica clássica, não apresenta uma maneira de construir o predicado, por isso essa lógica não é adequada para modelar computações.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p \vdash p}\, I}{\vdash p, \neg p}\, \neg_r}{\vdash p, p \vee \neg p}\, \vee_{r2}}{\vdash p \vee \neg p, p \vee \neg p}\, \vee_{r1}}{\vdash p \vee \neg p}\, C_r$$

Figura 3.2. Prova do princípio do terceiro excluído.

3.2 Lógica Intuicionista

Como foi dito anteriormente, a utilização da lógica para modelar computações só é interessante quando a lógica permite apenas provas construtivas, podendo assim relacionar os passos de uma prova com os passos de um algoritmo para se atingir um objetivo. A lógica intuicionista abandona a ideia de verdade absoluta da lógica clássica, e considera que uma afirmação é verdadeira se, e somente se, existe uma prova construtiva para a mesma. A sintaxe da lógica intuicionista é a mesma da lógica clássica (exceto pelo operador de implicação, representado por $\supset$ nessa lógica), a diferença crucial que irá permitir somente provas construtivas está no seu sistema de provas. A Figura 3.3 apresenta o sistema LJ em cálculo de sequentes para a lógica intuicionista. Nele os sequentes possuem no máximo uma fórmula do lado direito, e as regras de *weakening* e *contraction* não são permitidas nesse lado. É importante notar que, devido à essa restrição, o princípio do terceiro excluído não é provável nessa lógica.

REGRAS DE IDENTIDADE

$$\frac{}{P \longrightarrow P}\ [\mathrm{I}] \qquad \frac{\Gamma_1 \longrightarrow P \qquad \Gamma_2, P \longrightarrow C}{\Gamma_1, \Gamma_2 \longrightarrow C}\ [\mathrm{Cut}]$$

REGRAS LÓGICAS

$$\frac{}{\bot, \Gamma \longrightarrow \cdot}\ [\bot] \qquad \frac{}{\Gamma \longrightarrow \top}\ [\top]$$

$$\frac{P_i, \Gamma \longrightarrow C}{P_1 \wedge P_2, \Gamma \longrightarrow C}\ [\wedge_{li}] \qquad \frac{\Gamma \longrightarrow P \quad \Gamma \longrightarrow Q}{\Gamma \longrightarrow P \wedge Q}\ [\wedge_r]$$

$$\frac{P, \Gamma \longrightarrow C \quad Q, \Gamma \longrightarrow C}{P \vee Q, \Gamma \longrightarrow C}\ [\vee_l] \qquad \frac{\Gamma \longrightarrow P_i}{\Gamma \longrightarrow P_1 \vee P_2}\ [\vee_{ri}]$$

$$\frac{\Gamma_1 \longrightarrow P \quad Q, \Gamma_2 \longrightarrow C}{P \supset Q, \Gamma_1, \Gamma_2 \longrightarrow C}\ [\supset_l] \qquad \frac{\Gamma, P \longrightarrow Q}{\Gamma \longrightarrow P \supset Q}\ [\supset_r]$$

$$\frac{P[c/x], \Gamma \longrightarrow C}{\exists x.P, \Gamma \longrightarrow C}\ [\exists_l] \qquad \frac{\Gamma \longrightarrow P[t/x]}{\Gamma \longrightarrow \exists x.P}\ [\exists_r]$$

$$\frac{P[t/x], \Gamma \longrightarrow C}{\forall x.P, \Gamma \longrightarrow C}\ [\forall_l] \qquad \frac{\Gamma \longrightarrow P[c/x]}{\Gamma \longrightarrow \forall x.P}\ [\forall_r]$$

REGRAS ESTRUTURAIS

$$\frac{P, P, \Gamma \longrightarrow C}{P, \Gamma \longrightarrow C}\ [C_l]$$

$$\frac{\Gamma \longrightarrow C}{P, \Gamma \longrightarrow C}\ [W_l] \qquad \frac{\Gamma \longrightarrow \cdot}{\Gamma \longrightarrow P}\ [W_r]$$

Figura 3.3. Cálculo de sequentes para a lógica intuicionista. Nas regras $\forall_r$ e $\exists_l$, a variável c não ocorre livre em Γ ou C, e C representa uma fórmula ou nenhuma.

Mesmo apresentando somente provas construtivas, a utilização da lógica intuicionista como base para uma linguagem de programação depende de outros fatores. No caso da lógica ser utilizada para esse propósito, é necessário implementar um interpretador para executar os seus programas. Esses programas são descritos por fórmulas lógicas, e sua execução corresponde à prova de um objetivo (outra fórmula) utilizando o programa. Portanto, é necessário que a lógica implementada possua uma *estratégia uniforme de prova*, pelos motivos apresentados na Seção 1.3.

Infelizmente, a lógica intuicionista completa não apresenta uma estratégia uniforme de prova, de modo que as implementações existentes (as linguagens de programação Prolog e λ-Prolog) são de apenas subconjuntos da lógica. Nas seções seguintes esses fragmentos são apresentados, assim como suas restrições.

3.2.1 Cláusulas de Horn

As cláusulas de Horn são um subconjunto da lógica intuicionista que implementam a linguagem de programação lógica Prolog. Elas são descritas pela seguinte gramática:

$$\begin{array}{rcl} G & ::= & A \mid G \wedge G \\ P & ::= & A \mid G \supset A \mid \forall x.P \end{array}$$

Os objetivos (G) são formados apenas por átomos ou conjunções de átomos. Os programas (P) são compostos de átomos, objetivos implicando átomos ou o para todo ($\forall$). Com essa gramática é possível provar um subconjunto da lógica intuicionista utilizando uma estratégia uniforme de prova.

A estratégia utilizada para as cláusulas de Horn é chamada *backchaining*, e se baseia em tentativa e erro. Como o conjunto de possibilidades é muito pequeno, dada a gramática restrita, essa estratégia é aceitável. Suponha que seu objetivo G seja um átomo A. Se o programa $\mathcal{P}$ contém a fórmula A, então temos uma prova óbvia e a computação termina. Se $\mathcal{P}$ contém uma cláusula da forma $G' \supset A$, então sabemos que, se conseguimos provar G', temos também uma prova para A. Logo, o problema se reduz a provar $\mathcal{P} \rightarrow G'$. Caso se desenvolva G' e não chegue a lugar nenhum, é necessário voltar ao ponto de escolha de $G' \supset A$ e tentar outra cláusula do programa. Caso todas as cláusulas sejam testadas em uma iteração e não se consiga uma prova, não é possível provar o objetivo a partir do programa dado. Se o objetivo G for uma conjunção de átomos, basta tentar provar cada um deles separadamente utilizando essa estratégia. Seu objetivo será provado se for obtida uma prova para cada átomo.

É importante observar que, durante a busca por uma prova, todos os objetivos serão derivados do mesmo programa, e todas as cláusulas utilizadas para a prova serão extraídas também desse mesmo programa. Isso significa que os sequentes não podem ser muito alterados durante a derivação (o lado esquerdo do sequente, em particular, é sempre estático) e a lógica em si tem uma influência direta muito pequena sobre o processamento.

Note também que todas as cláusulas que serão necessárias para provar qualquer objetivo devem estar no programa desde o início da computação. Isso não oferece liberdade para modularização de programas ou abstração de dados; todos os dados necessários devem estar explícitos desde o início em um grande e único bloco. Esse

tipo de restrição é uma das motivações para tentar ir além das cláusulas de Horn.

3.2.2 Fórmulas de Harrop

A gramática que gera as cláusulas de Horn pode ser expandida para cobrir um conjunto maior de fórmulas da lógica intuicionista, sem perder a propriedade de estratégia uniforme de prova. Essa nova gramática gera as fórmulas de Harrop de primeira ordem, que são implementadas na linguagem de programação lógica λ-Prolog, e sua descrição mais comum é a seguinte:

$$\begin{array}{rcl} G & ::= & A \mid G \wedge G \mid P \supset G \mid \forall x.G \\ P & ::= & A \mid G \supset A \mid \forall x.P \end{array}$$

A diferença dessa gramática para a gramática das cláusulas de Horn é a adição da implicação ($P \supset G$) e do "para todo" ($\forall x.G$) no objetivo. A utilização da implicação no objetivo permite a modularização de um programa, e a utilização do "para todo" no objetivo permite abstração de dados no programa [Miller, 1989].

Dessa forma, as propriedades de modularização e abstração de dados são provenientes da própria especificação lógica.

A busca por provas é realizada também utilizando *backchaining*. A grande diferença é que, nas fórmulas de Harrop, lado esquerdo do sequente (programa P) pode ser aumentado durante uma derivação (mas nunca diminuído).

As fórmulas de Harrop são um conjunto maximal da lógica intuicionista que possui uma estratégia uniforme de prova, e mesmo assim, é ainda muito restrito. Essa é a principal motivação para se procurar outras lógicas para representar computações.

3.3 Lógica Linear

A lógica linear, proposta por Girard [Girard, 1987], pode ser vista como um refinamento da lógica clássica. Sua sintaxe é descrita pela seguinte gramática:

$$\begin{array}{rl} P ::= & A \mid P \otimes P \mid P \oplus P \mid 1 \mid 0 \mid !P \mid \exists x.P \mid A^{\perp} \mid \\ & P \& P \mid P \parr P \mid \perp \mid \top \mid ?P \mid \forall x.P \end{array}$$

Essa é uma lógica de recursos finitos, ou seja, as fórmulas são consumidas quando utilizadas. A motivação para se estudar uma lógica desse tipo é sua maior aproximação da realidade, já que, em geral, os recursos não são infinitos. Considere por exemplo a modelagem de uma máquina de estados finitos que representa uma máquina de refrigerantes. Ela pode ser descrita por uma fórmula lógica simples: $\$2 \Rightarrow refrigerante$. Porém, o dinheiro é um recurso finito, e uma vez que essa regra é aplicada, a quantidade de dinheiro disponível é diminuída em duas unidades. Portanto, essa operação seria melhor representada pela implicação linear: $\$2 \multimap refrigerante$, cuja semântica deixa explícito o consumo de duas unidades monetárias para a aquisição de um refrigerante.

$$\frac{\Delta \vdash \Gamma}{\Delta, !B \vdash \Gamma}\ !W \qquad \frac{\Delta, !B, !B \vdash \Gamma}{\Delta, !B \vdash \Gamma}\ !C \qquad \frac{\Delta, B \vdash \Gamma}{\Delta, !B \vdash \Gamma}\ !D \qquad \frac{!\Delta \vdash B, ?\Gamma}{!\Delta \vdash !B, ?\Gamma}\ !R$$

$$\frac{\Delta \vdash \Gamma}{\Delta \vdash ?B, \Gamma}\ ?W \qquad \frac{\Delta \vdash ?B, ?B\Gamma}{\Delta \vdash ?B, \Gamma}\ ?C \qquad \frac{\Delta \vdash B, \Gamma}{\Delta \vdash ?B, \Gamma}\ ?D \qquad \frac{!\Delta, B \vdash ?\Gamma}{!\Delta, ?B \vdash ?\Gamma}\ ?L$$

Figura 3.4. Regras de inferência para os operadores exponenciais.

Observe que não incluímos na sintaxe a implicação ($\multimap$). De fato, identificamos $A \multimap B$ com $A^{\perp} \parr B$.

Mesmo sendo uma lógica de recursos finitos, é desejável ainda que algumas fórmulas possuam comportamento "clássico", i.e., possam ser utilizadas sempre que necessário. Essas fórmulas são marcadas com os operadores ! (para fórmulas do lado esquerdo do sequente) ou ? (para fórmulas do lado direito do sequente), chamados **exponenciais**. Fórmulas com esse marcador podem sofrer *contraction* e *weakening*. Devido a essa distinção de tipos de fórmulas, o contexto de sequentes na lógica linear é frequentemente dividido em dois conjuntos: as fórmulas clássicas (um conjunto) e as fórmulas lineares (um multiconjunto). No exemplo da máquina de refrigerantes dado acima, a fórmula $\$2 \multimap refrigerante$ é sempre verdadeira (é uma fórmula clássica), apesar de ela só poder ser aplicada se houver de fato a quantidade de dinheiro suficiente (a existência de duas unidades monetárias é uma fórmula linear). As regras envolvendo fórmulas com exponenciais são mostradas na Figura 3.4.

A partir das regras $!W$ e $!C$ é possível perceber que fórmulas com o exponencial ! à esquerda têm comportamento clássico. O mesmo ocorre com as fórmulas com o exponencial ? à direita, como mostram as regras $?W$ e $?C$. As regras $!D$ e $?D$ transformam uma fórmula clássica em linear, ou seja, a partir desse ponto da prova em diante, a fórmula B pode ser consumida e não poderá ser mais utilizada. As regras $!R$ e $?L$ são mais interessantes. Para retirar os exponenciais, todas as outras fórmulas devem ter comportamento clássico, i.e., não podem haver fórmulas lineares no sequente[1]. Isso significa que o contexto linear deve estar vazio. Esse tipo de controle sobre o contexto torna possível a verificação da vacuidade de um subconjunto de fórmulas do sequente e permite que alguns sistemas sejam especificados com essa lógica de modo mais natural.

Outra diferença da lógica clássica é a existência de duas versões, **aditiva** e **multiplicativa**, para os operadores de conjunção ($\wedge$) e disjunção ($\vee$). As conjunções são representadas pelos conectivos: $\&$ (aditiva) e $\otimes$ (multiplicativa). As disjunções são representadas pelos conectivos: $\oplus$ (aditiva) e $\parr$ (multiplicativa). A principal diferença entre essas duas versões dos operadores é o modo como os contextos da conclusão e da premissa são tratados. Na versão aditiva, o contexto da conclusão é igual ao contexto das premissas, enquanto na versão multiplicativa, o contexto da conclusão é a união dos contextos das premissas. Se existissem essas duas versões da conjunção na lógica clássica, elas seriam descritas pelas seguintes regras de inferência ($\wedge_a$ é a versão aditiva e $\wedge_m$ é a versão multiplicativa):

[1] Nos sequentes das regras $!R$ e $?L$, $!\Delta$ significa a aplicação de ! para todas as fórmulas do conjunto Δ e $?\Gamma$ é a aplicação de ? a todas as fórmulas do conjunto Γ.

$$\frac{\vdash \Delta, A \qquad \vdash \Delta, B}{\vdash \Delta, A \wedge_a B}\,[\wedge_a] \qquad \frac{\vdash \Delta_1, A \qquad \vdash \Delta_2, B}{\vdash \Delta_1, \Delta_2, A \wedge_m B}\,[\wedge_m]$$

Porém isso não é necessário, já que é possível obter o comportamento de uma regra utilizando a outra, como é mostrado a seguir.

- A conjunção multiplicativa pode ser obtida através da conjunção aditiva utilizando regras de *weakening* para apagar do contexto algumas fórmulas, de maneira a se ter dois subconjuntos disjuntos Δ_1 e Δ_2 cuja união é o contexto original Δ.

$$\frac{\dfrac{\vdash \Delta_1, A}{\vdash \Delta, A}\,[n \times W] \qquad \dfrac{\vdash \Delta_2, B}{\vdash \Delta, B}\,[m \times W]}{\vdash \Delta, A \wedge_a B}\,[\wedge_a]$$

- A conjunção aditiva pode ser obtida através da conjunção multiplicativa utilizando regras de *contraction* para duplicar os elementos do contexto de maneira que quando a divisão do mesmo for realizada, os dois subconjuntos sejam iguais ao contexto original.

$$\frac{\dfrac{\vdash \Delta_1, \Delta_2, A \qquad \vdash \Delta_1, \Delta_2, B}{\vdash \Delta_1, \Delta_1, \Delta_2, \Delta_2 A \wedge_m B}\,[\wedge_m]}{\vdash \Delta_1, \Delta_2, A \wedge_m B}\,[n \times C]$$

Essas transformações são possíveis porque as operações de *weakening* e *contraction* podem ser aplicadas em qualquer fórmula. Quando estamos trabalhando com recursos finitos, essas operações não podem ser utilizadas indiscriminadamente e portanto é necessário diferenciar dois operadores de conjunção e disjunção. Devido a essa separação, a lógica linear também apresenta elementos neutros distintos para cada um dos operadores. Eles são 1, 0, $\top$ e $\bot$, e as seguintes equivalências são válidas:

$$\bot \parr A \equiv A \qquad 1 \otimes A \equiv A$$
$$\top \& A \equiv A \qquad 0 \oplus A \equiv A$$

A prova do princípio do terceiro excluído na lógica linear é um bom exemplo de como funcionam os exponenciais. A disjunção correspondente àquela utilizada na lógica clássica e intuicionista é a aditiva, já que nas duas lógicas não existe divisão de contexto. Portanto, a fórmula $a \oplus \neg a$ corresponde à $a \vee \neg a$. Ao tentar provar essa fórmula na lógica linear, é encontrado o mesmo problema da lógica intuicionista:

$$\frac{\dfrac{?}{\vdash a}}{\vdash a \oplus \neg a}\,\oplus_1 \qquad \frac{\dfrac{?}{\vdash \neg a}}{\vdash a \oplus \neg a}\,\oplus_2$$

Em ambos os casos chega-se a um sequente que não é provável. Porém, ao utilizar o exponencial ? para que essa fórmula tenha um comportamento clássico, a prova é

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash a, \neg a}\, I}{\vdash a, a \oplus \neg a}\, \oplus}{\vdash a \oplus \neg a, a \oplus \neg a}\, \oplus}{\vdash ?(a \oplus \neg a), ?(a \oplus \neg a)}\, ?D}{\vdash ?(a \oplus \neg a)}\, ?C$$

Figura 3.5. Prova do princípio do terceiro excluído na lógica linear.

$$\begin{array}{rlrl}
(P \otimes Q)^{\perp} &\equiv P^{\perp} \parr Q^{\perp} & (P \parr Q)^{\perp} &\equiv P^{\perp} \otimes Q^{\perp} \\
(P \& Q)^{\perp} &\equiv P^{\perp} \oplus Q^{\perp} & (P \oplus Q)^{\perp} &\equiv P^{\perp} \& Q^{\perp} \\
(\exists x.P)^{\perp} &\equiv \forall x.P^{\perp} & (\forall x.P)^{\perp} &\equiv \exists x.P^{\perp} \\
(?P)^{\perp} &\equiv !P^{\perp} & (!P)^{\perp} &\equiv ?P^{\perp}
\end{array}$$

Figura 3.6. Leis de DeMorgan para a lógica linear.

imediata, como mostra a Figura 3.5. De fato, o princípio do terceiro excluído é provável na lógica clássica.

É importante notar que a negação na lógica linear é involutiva, i.e., $A^{\perp\perp} \equiv A$. E também que os conectivos são duais, dois a dois. Devido a esse fato, a lógica linear possui uma versão *one-sided*, na qual os sequentes apresentam fórmulas apenas do lado direito. As fórmulas do lado esquerdo são negadas, de forma que a negação é aplicada utilizando as leis de De Morgan, de acordo com as equivalências da Figura 3.6, até ter escopo atômico. De fato, na gramática apresentada, a negação é aplicada somente a átomos (A), e não a fórmulas (P).

O sistema de cálculo de sequentes para a lógica linear, apresentado na Figura 3.7, é *one-sided*, dado que todas as fórmulas estão do lado direito de $\vdash$, e é também diádico[2], já que as fórmulas são divididas em dois conjuntos, Θ e Γ (o contexto clássico e o contexto linear, respectivamente). Nessa presentação, todas as fórmulas marcadas com ? são colocadas no conjunto Θ sem o marcador, e sabe-se que fórmulas desse conjunto podem sofrer *contraction* e *weakening* à vontade. As outras fórmulas estão no conjunto Γ, e, uma vez utilizadas, são consumidas. Todas as regras de inferência são aplicadas a fórmulas de Γ, o que não causa problemas já que as fórmulas de Θ podem ser copiadas para Γ quando for necessário com a regra $[D?]$. A regra [!] também apresenta o mesmo comportamento já que exige que, para que o ! seja retirado, o contexto linear não contenha fórmula alguma além daquela marcada. Finalmente, pode ser observado na regra *Cut* que fórmulas que deveriam estar no lado esquerdo aparecem negadas do lado direito.

[2]Formado por duas partes.

REGRAS DE IDENTIDADE

$$\frac{}{\vdash \Theta : P, P^{\perp}}\ [\mathrm{I}] \qquad \frac{\vdash \Theta : \Gamma, P^{\perp} \qquad \vdash \Theta : \Delta, P}{\vdash \Theta : \Gamma, \Delta}\ [\mathrm{Cut}]$$

REGRAS LÓGICAS

$$\frac{}{\vdash \Theta : 1}\ [1] \qquad \frac{}{\vdash \Theta : \Gamma, \top}\ [\top] \qquad \frac{\vdash \Theta : \Gamma}{\vdash \Theta : \Gamma, \bot}\ [\bot]$$

$$\frac{\vdash \Theta : \Gamma, P_i}{\vdash \Theta : \Gamma, P_1 \oplus P_2}\ [\oplus_i] \qquad \frac{\vdash \Theta : \Gamma, P \quad \vdash \Theta : \Gamma, Q}{\vdash \Theta : \Gamma, P \& Q}\ [\&]$$

$$\frac{\vdash \Theta : \Gamma, P, Q}{\vdash \Theta : \Gamma, P \parr Q}\ [\parr] \qquad \frac{\vdash \Theta : \Gamma, P \quad \vdash \Theta : \Delta, Q}{\vdash \Theta : \Gamma, \Delta, P \otimes Q}\ [\otimes]$$

$$\frac{\vdash \Theta : \Gamma, P[t/x]}{\vdash \Theta : \Gamma, \exists x.P}\ [\exists] \qquad \frac{\vdash \Theta : \Gamma, P[c/x]}{\vdash \Theta : \Gamma, \forall x.P}\ [\forall]$$

$$\frac{\vdash \Theta, P : \Gamma, P}{\vdash \Theta, P : \Gamma}\ [\mathrm{D?}] \qquad \frac{\vdash \Theta : P}{\vdash \Theta : !P}\ [!]$$

REGRAS ESTRUTURAIS

$$\frac{\vdash \Theta, P : \Gamma}{\vdash \Theta : \Gamma, ?P}\ [?]$$

Figura 3.7. Sistema de cálculo de sequentes diádicos e *one-sided* para a lógica linear. Na regra [$\forall$], a variável c não aparece livre em Γ.

Capítulo 4

Lógica Linear com Subexponenciais

4.1 Lógica Linear com Subexponenciais

4.1.1 Sintaxe

A sintaxe da lógica linear com subexponenciais é descrita pela seguinte gramática:

$$\begin{aligned} P ::= & A \mid P \otimes P \mid P \oplus P \mid 1 \mid 0 \mid !^i P \mid \exists x.P \mid A^{\perp} \mid \\ & P \& P \mid P \parr P \mid \bot \mid \top \mid ?^i P \mid \forall x.P \mid \Cup_l loc.P \mid \Cap_l loc.P \end{aligned}$$

Ela é um refinamento da lógica linear, apresentando os novos operadores: $!^i$, $?^i$, $\Cup_l$ e $\Cap_l$. Observe que novamente não está incluída na sintaxe a implicação ($\multimap$). De fato, identifica-se $A \multimap B$ com $A^{\perp} \parr B$. Observe também que a negação ($\perp$) é aplicada somente a átomos (A), e não a fórmulas (P). Isso significa que todas as fórmulas estão na sua forma normal de negação. O que é possível porque a negação é involutiva.

Os conectivos são parecidos com aqueles da lógica linear: $\otimes$ (conjunção) e $\parr$ (disjunção), e suas unidades 1 e $\perp$, são multiplicativos; $\oplus$ (disjunção) e $\&$ (conjunção), e suas unidades 0 e $\top$, são aditivos; $\forall$ e $\exists$ são quantificadores; $!^i$ e $?^i$ são chamados **subexponenciais**, e $\Cup_l$ e $\Cap_l$ são utilizados para instanciar e criar novos subexponenciais (são como quantificadores, porém, atuam em índices subexponenciais ao invés de variáveis).

O sistema de cálculo de sequentes para a lógica linear com subexponenciais é apresentado na Figura 4.7.

4.1.2 Subexponenciais

Na lógica linear, as fórmulas do contexto de um sequente podem ser divididas em dois grupos: as "clássicas" (que podem sofrer *contraction* e *weakening*) e as "lineares" (aquelas que não podem sofrer nem *contraction* e nem *weakening* e são portanto consumidas quando são utilizadas).

$$\frac{\Gamma \vdash \Delta, A \quad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \wedge^1 B} \wedge^1_r \qquad \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge^1 B \vdash \Delta} \wedge^1_l \qquad \frac{\Gamma \vdash \Delta, A \quad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \wedge^2 B} \wedge^2_r \qquad \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge^2 B \vdash \Delta} \wedge^2_l$$

Figura 4.1. Regras de inferência para dois tipos de conjunção.

Ao utilizar uma lógica como meta-lógica para especificação de outros sistemas, as restrições estruturais do sistema objeto devem ser corretamente capturadas na meta-lógica. Para que isso fique mais claro, considere que o sistema a ser especificado é o cálculo de sequentes para a lógica intuicionista, descrito na Figura 3.3. Nele, as fórmulas do lado esquerdo do sequente podem sofrer *contraction* e *weakening* à vontade. Porém, no lado direito do sequente deve haver sempre uma ou nenhuma fórmula. Pode-se então considerar que o lado esquerdo é formado por um conjunto de fórmulas e o lado direito é um multiconjunto[1]. Ao especificar esse sistema em lógica linear, é natural considerar que as fórmulas do lado esquerdo do sequente estarão no contexto clássico, e a fórmula do lado direito estará no contexto linear. O *weakening* dessa fórmula, permitido pela regra W_r, será especificado por uma fórmula em LL, como será mostrado na Seção 5.2.

Porém, a existência de apenas dois contextos distintos, um como um conjunto e outro como multiconjunto, torna limitada a expressividade dessa lógica como *framework* lógico. A especificação de sistemas cujos elementos devem ser divididos em dois conjuntos, dois multiconjuntos, ou mais do que duas partições, não é possível ser feita de maneira tão direta. A solução para esse problema foi encontrada nos operadores *subexponenciais*. Com eles, é possível dividir o contexto de um sequente em lógica linear em quantos conjuntos se queira, e atribuir qualquer uma das seguintes regras para cada um desses conjuntos:

- todas as fórmulas sofrem *contraction* e *weakening*;
- nenhuma fórmula sofre *contraction* nem *weakening*;
- todas as fórmulas sofrem apenas *contraction*;
- todas as fórmulas sofrem apenas *weakening*.

Os operadores subexponenciais são a divisão dos operadores exponenciais, ? e !, em "classes". Isso é possível porque estes são operadores não canônicos. Quando um operador é canônico, como por exemplo a conjunção ($\wedge$), não é possível definir tipos diferentes do mesmo. Suponha que sejam definidas as regras para dois tipos de conjunção ($\wedge^1$ e $\wedge^2$) da Figura 4.1.

Com elas, é possível provar que, para quaisquer fórmulas A e B, $A \wedge^1 B \equiv A \wedge^2 B$. Para isso, basta apenas mostrar que $A \wedge^1 B \vdash A \wedge^2 B$ e $A \wedge^2 B \vdash A \wedge^1 B$, o que é feito pela prova da Figura 4.2. Isso significa que a provabilidade de uma fórmula independe do tipo de conjunção utilizada.

[1]Em um conjunto, elementos duplicados não são considerados, ou seja $\{a\} \equiv \{a, a\}$. Essa propriedade não é válida para multiconjuntos.

$$\dfrac{\dfrac{\dfrac{}{A, B \vdash A}\, I}{A \wedge^i B \vdash A}\, \wedge_l^i \quad \dfrac{\dfrac{}{A, B \vdash B}\, I}{A \wedge^i B \vdash B}\, \wedge_l^i}{A \wedge^i B \vdash A \wedge^j B}\, \wedge_r^j$$

Figura 4.2. Prova da equivalência entre os dois tipos de conjunção.

$$\dfrac{\vdash ?^1\Gamma, F}{\vdash ?^1\Gamma, !^1F}\, !^1 \quad \dfrac{\vdash \Gamma, F}{\vdash \Gamma, ?^1F}\, ?^1D \quad \dfrac{\vdash ?^2\Gamma, F}{\vdash ?^2\Gamma, !^2F}\, !^2 \quad \dfrac{\vdash \Gamma, F}{\vdash \Gamma, ?^2F}\, ?^2D$$

Figura 4.3. Regras de inferência para os dois tipos de exponenciais.

O mesmo não ocorre com os exponenciais. Suponha que sejam criados dois tipos de exponenciais, $!^1$, $?^1$ e $!^2$, $?^2$, cujas regras de inferência são mostradas na Figura 4.3.

Dadas essas novas regras, observe o que ocorre quando tentamos provar a equivalência $!^1F \equiv !^2F$:

$$\dfrac{\dfrac{?}{\vdash !^2F, ?^1F}}{\vdash !^2F, (!^1F)^\perp}$$

Nesse ponto não podemos aplicar regra alguma, portanto $!^1F \neq !^2F$. Isso significa que é possível definir quantos operadores subexponenciais se queira. Cada um deles é caracterizado pelo seu índice i, e eles seguem uma ordem parcial pré-definida ($\preceq$). Essa ordenação será utilizada na aplicação da regra $!^i$, como pode ser visto na Figura 4.7. A hierarquia dos subexponenciais define aquelas fórmulas que serão apagadas ao se aplicar essa regra de inferência.

O sequente da lógica linear *one-sided* com subexponenciais pode ser representado por:

$$\vdash ?^{i_1}\Theta_1, ?^{i_2}\Theta_2, ..., ?^{i_n}\Theta_n, \Gamma$$

Cada conjunto Θ_k contém aquelas fórmulas que possuem o subexponencial $?^{i_k}$ como conectivo principal. As fórmulas em Γ são aquelas que têm comportamento linear. Cada conjunto de fórmulas definido por um índice subexponencial é chamado *location*, já que funciona como um "repositório" de fórmulas cujo operador mais externo é o mesmo subexponencial. Além disso, é possível definir quais regras estruturais (*contraction* e *weakening*) podem ser aplicadas para quais índices subexponenciais. Com esse maior controle sobre o contexto, é possível inserir e retirar fórmulas de *locations* específicos e verificar se um *location* está vazio ou não, como mostra a Seção 4.2.1.

O contexto do sequente da lógica linear com subexponenciais é formado por uma lista de conjuntos, denotada pela letra $\mathcal{K}$. Ela é indexada pelos índices subexponenciais existentes e a posição $\mathcal{K}[i_j]$ armazena todas as fórmulas cujo conectivo principal é $?^{i_j}$. Além dessa lista, também está presente no sequente o conjunto Γ de fórmulas lineares.

Com essa nova configuração, para cada sistema será definida uma *assinatura subexponencial*: $\Sigma = \langle \mathcal{I}, \preceq, \mathcal{W}, \mathcal{C} \rangle$. $\mathcal{I}$ contém os índices subexponenciais utilizados, $\preceq$

- $(\mathcal{K}_1 \otimes \mathcal{K}_2)[i] = \begin{cases} \mathcal{K}_1[i] \cup \mathcal{K}_2[i] & \text{se } i \notin \mathcal{C} \\ \mathcal{K}_1[i] & \text{se } i \in \mathcal{C} \cap \mathcal{W} \end{cases}$
- $\mathcal{K}[\mathcal{S}] = \bigcup\{\mathcal{K}[i] \mid i \in \mathcal{S}\}$
- $(\mathcal{K} +_l A)[i] = \begin{cases} \mathcal{K}[i] \cup \{A\} & \text{se } i = l \\ \mathcal{K}[i] & \text{c.c.} \end{cases}$
- $\mathcal{K} \leq_i [l] = \begin{cases} \mathcal{K}[l] & \text{se } i \preceq l \\ \emptyset & \text{se } i \not\preceq l \end{cases}$
- $(\mathcal{K}_1 \star \mathcal{K}_2) \mid_{\mathcal{S}}$ verdadeiro se, e somente se, $(\mathcal{K}_1[j] \star \mathcal{K}_2[j])$

Figura 4.4. Especificação das operações sobre o contexto. Nessa figura, $i \in I$, $j \in \mathcal{S}$, $\mathcal{S} \subseteq I$, e o conectivo binário $\star \in \{=, \subset, \subseteq\}$.

é a relação de pré-ordem entre os índices, $\mathcal{C}$ e $\mathcal{W}$ contém os índices que podem sofrer *contraction* e que podem sofrer *weakening*, respectivamente. Além disso, as operações com o contexto em algumas regras foram redefinidas. As novas operações são definidas na Figura 4.4.

4.2 *Locations*

4.2.1 Verificando se um *location* é vazio

Uma das principais vantagens de se ter os subexponenciais na lógica linear é a possibilidade de verificar a existência de certas estruturas de dados (objetos) em um *location* específico. Para isso, podemos utilizar a regra $!^l$. Essa operação é explicada com detalhes nessa seção.

Primeiramente, observe a regra $!^l$:

$$\frac{\vdash \mathcal{K} \leq_l : \cdot \Uparrow A}{\vdash \mathcal{K} : \cdot \Downarrow !^l A} \; [!^l, \text{ dado que } \mathcal{K}[\{x \mid l \not\preceq x \wedge x \notin \mathcal{W}\}] = \emptyset]$$

$\mathcal{K}_{\leq l}$ é definido como todos os conjuntos $\mathcal{K}[i]$ tal que $l \preceq i$, ou seja, todos os índices subexponenciais menores que ou não relacionados a l terão seu conjunto esvaziado (as fórmulas serão apagadas). A condição $\mathcal{K}[\{x \mid l \not\preceq x \wedge x \notin \mathcal{W}\}] = \emptyset$ dessa regra garante que apenas as fórmulas que possam sofrer *weakening* sejam apagadas. A prova da fórmula A continua somente se essa condição for garantida. Dessa maneira, tem-se um modo de verificar se alguns índices estão vazios ou não, mais especificamente, os índices que são menores que ou não relacionados a l e que não podem sofrer *weakening*.

Para que a verificação se restrinja a apenas um índice (e não a todos aqueles menores que ou não relacionados a l), algumas manipulações devem ser feitas. A assinatura subexponencial é estendida com os seguintes elementos:

- para cada índice l da assinatura original, cria-se um índice $\hat{l}$ ($\mathcal{K}[\hat{l}] = \emptyset$ durante toda a prova);
- é incluída a relação $\hat{l} \preceq k$ para todo $k \neq l$ da assinatura original.

Dessa forma, não existem índices menores que $\hat{l}$, e o único não relacionado a ele será l. Portanto, a aplicação da regra $!^{\hat{l}}$ irá verificar se apenas o índice subexponencial l está vazio[2].

4.2.2 Instanciação e criação de *locations*

Locations podem ser instanciados e criados utilizando as regras de inferência a seguir. Ambas apresentam o conjunto de *locations* $\mathcal{L}$ explícito.

$$\frac{\mathcal{L} \vdash \mathcal{K}, C[s/loc, \hat{s}/\hat{loc}]}{\mathcal{L} \vdash \mathcal{K}, \Cup_l loc.C} \; [\Cup_l, \text{dado que } s \in \mathcal{L}]$$

Essa regra instancia os *locations* da fórmula C com um *location* s já existente. Ela pode ser vista como uma regra $\exists$ que instancia *locations* ao invés de variáveis.

$$\frac{\mathcal{L} \cup loc \vdash \mathcal{K}, C}{\mathcal{L} \vdash \mathcal{K}, \Cap_l loc.C} \; [\Cap_l, \text{dado que } loc \text{ é um novo location}]$$

Essa regra cria um novo *location* loc e o adiciona ao conjunto $\mathcal{L}$. Ela é análoga à regra $\forall$ para variáveis.

Esse novo *location* é completamente linear (não pode sofrer *contraction* ou *weakening*) e não estará relacionado a nenhum dos outros pela ordem de precedência, exceto do location ∞ do qual todos são menores.

4.3 *Focusing*

Um dos problemas encontrados na implementação de interpretadores para uma lógica é o não-determinismo na aplicação de regras de inferência. A escolha "cega" de qual regra deve ser aplicada em cada passo pode levar a uma busca redundante ou até mesmo a *loops* infinitos. No primeiro caso será realizado um trabalho desnecessário, enquanto no segundo caso a derivabilidade ou não de uma fórmula pode nunca ser determinada. De qualquer maneira, essas são duas situações que se deseja evitar. Isso é feito definindo uma estratégia uniforme de prova, que é nada mais do que uma disciplina para escolher qual a próxima regra de inferência a ser aplicada em um sequente.

Para resolver o problema da busca por provas na lógica linear, Andreoli desenvolveu uma disciplina de busca por provas chamada *focusing* [Andreoli, 1992]. Ele propôs o sistema de provas focado[3] para a lógica linear e provou que ele é completo, i.e., qualquer prova em lógica linear pode ser representada por uma prova focada. Esse tipo de prova pode ser visto como a normalização de um conjunto de provas da lógica linear que são equivalentes, no sentido que uma pode ser obtida da outra pela permutação de aplicação de regras e eliminação ou introdução de loops inúteis. Uma prova focada envolve a aplicação de regras de inferência em duas fases alternantes. Para isso,

[2] A verificação só faz sentido se l não puder sofrer *weakening*, já que, caso contrário, as fórmulas poderiam ser apagadas quando fosse necessário.

[3] do inglês *focused*

Assíncronos	Síncronos
$\parr$	$\otimes$
$\&$	$\oplus$
?	!
$\top$	1
$\bot$	0
$\forall$	$\exists$

Tabela 4.1. Divisão dos operadores da lógica linear em assíncronos e síncronos.

os conectivos da lógica linear são divididos em **síncronos** e **assíncronos**, como mostra a Tabela 4.1.

Dessa maneira, podemos também classificar uma fórmula como síncrona (positiva) ou assíncrona (negativa). Fórmulas positivas têm um conectivo síncrono como seu conectivo mais externo, enquanto fórmulas negativas têm um conectivo assíncrono como seu conectivo mais externo.

Essa divisão é baseada nas seguintes propriedades dos dois grupos: seja F a fórmula focada (aquela escolhida para se aplicar a regra de inferência). Se F é uma fórmula assíncrona, então existe apenas uma instância da regra de inferência do seu conectivo (assíncrono) para ser aplicada. Se F é uma fórmula síncrona, então existem zero ou mais instâncias da regra de inferência do seu conectivo (síncrono) para ser aplicada.

A existência de várias instâncias de uma regra pode ser observada, por exemplo, na regra $[\otimes]$, na qual cada instância corresponde a um particionamento diferente do conjunto de fórmulas $\mathcal{K}$. Isso significa que podem ser necessárias várias tentativas de aplicação dessa regra até que o particionamento correto seja encontrado, ou seja, pode ser necessário fazer *backtracking*. Note que as regras de inferência para os conectivos assíncronos apresentam somente uma maneira de serem aplicadas.

Outro modo interessante para caracterizar a diferença entre conectivos síncronos e assíncronos é a invertibilidade das regras de inferência. Em uma regra de um conectivo assíncrono, a conclusão (parte inferior) é válida **se, e somente se**, as premissas (parte superior) forem válidas. Para as regras dos conectivos síncronos isso não é necessariamente verdadeiro. Tome como exemplo a regra $[\oplus_1]$, da disjunção aditiva: se a premissa A_i for verdade, certamente $A_1 \oplus A_2$ será verdade, porém a volta ($A_1 \oplus A_2 \Rightarrow A_i$, $i \in \{1, 2\}$) é obviamente inválida.

Provas focadas são formadas por duas fases alternantes: assíncrona e síncrona. Uma fase assíncrona é composta por aplicações de regras de inferência de conectivos assíncronos, e uma fase síncrona é composta por aplicação de regras de inferência de conectivos síncronos. Essas regras são mostradas na Figura 4.7. O sequente do sistema de provas focado para a lógica linear clássica é formado por três elementos: um conjunto Θ de fórmulas clássicas, um multiconjunto Γ de fórmulas lineares e o objetivo (*goal*). Esse objetivo é exatamente uma fórmula na fase síncrona, e pode ser nenhuma ou mais fórmulas na fase assíncrona. Além desses elementos, existe uma seta vertical em cada sequente que indica uma fase assíncrona ($\Uparrow$) ou uma fase síncrona ($\Downarrow$). A lógica linear

com subexponenciais possui os mesmos elementos, com a diferença que Θ é substituído pelo conjunto de *locations* $\mathcal{K}$:

$$\mathcal{K} : \Gamma \Uparrow L \quad \mathcal{K} : \Gamma \Downarrow F$$

As fórmulas do lado direito dessa seta são ditas "focadas"[4], e regras de inferência são aplicadas somente a essas fórmulas seguindo a seguinte disciplina: aplicam-se regras assíncronas sempre que possível, passando as fórmulas síncronas para o contexto (regra [R⇑]), e uma fase síncrona é iniciada somente se não existirem mais fórmulas negativas no objetivo, uma fórmula positiva é escolhida do contexto com uma das regras $[D_l]$ ou $[D_1]$. Além disso, se uma fórmula negativa for encontrada enquanto se decompõe uma fórmula positiva, a fase síncrona é imediatamente interrompida e inicia-se uma fase assíncrona (regra [R⇓]) até que a fórmula negativa seja completamente decomposta.

Considere, por exemplo, a prova da equivalência $(P \oplus Q)^\perp \equiv P^\perp \& Q^\perp$ nas Figuras 4.5 e 4.6. Como o sistema é *one-sided*, a fórmula $(P \oplus Q)^\perp$ é passada para o lado direito sem a negação. Se a primeira regra a ser aplicada for $[\oplus]$, não é possível encontrar uma prova para o sequente, como mostra a Figura 4.5. Porém, encontra-se a prova ao iniciar por [&]. Utilizando a estratégia de *focusing*, a primeira regra a ser aplicada seria [&] a $P^\perp \& Q^\perp$, por se tratar de uma fórmula assíncrona.

$$\dfrac{\dfrac{\dfrac{}{\vdash P, P^\perp}\, I \quad \dfrac{?}{\vdash P, Q^\perp}}{\vdash P, P^\perp \& Q^\perp}\, \&}{\vdash P \oplus Q, P^\perp \& Q^\perp}\, \oplus_1$$

Figura 4.5. Prova sem utilizar *focusing*.

$$\dfrac{\dfrac{\dfrac{}{\vdash P, P^\perp}\, I}{\vdash P \oplus Q, P^\perp}\, \oplus_1 \quad \dfrac{\dfrac{}{\vdash Q, Q^\perp}\, I}{\vdash P \oplus Q, Q^\perp}\, \oplus_2}{\vdash P \oplus Q, P^\perp \& Q^\perp}\, \&$$

Figura 4.6. Prova utilizando *focusing*.

4.4 Operadores *delay*

Uma prova utilizando *focusing* alterna entre duas fases: síncrona (ou positiva) e assíncrona (ou negativa). Essa divisão de fases foi proposta por Andreoli [Andreoli, 1992] para definir uma estratégia de aplicação das regras de inferência e reduzir o espaço de busca, eliminando provas equivalentes, como foi explicado na Seção 4.3.

[4] É importante observar que em uma fase assíncrona é possível focar em um conjunto de fórmulas, enquanto em uma fase síncrona foca-se em apenas uma fórmula.

A polaridade de uma fórmula (positiva ou negativa) é determinada por seu conectivo mais externo, e uma fase negativa ou positiva da prova de um sequente é caracterizada pela aplicação consecutiva de regras de inferência de conectivos negativos ou positivos, respectivamente.

Algumas vezes, durante uma prova, pode ser interessante parar uma fase (positiva ou negativa) para que ela não se torne muito grande. Isso pode ser útil, por exemplo, ao se definir conectivos sintéticos, que são como "macros" para sequências de aplicações de regras utilizadas com frequência. Para se conseguir alternar entre fases artificialmente são utilizados "operadores" chamados *delays* (atrasos). Um *delay* pode ser negativo (força a prova a entrar em uma fase negativa, independente da anterior) ou positivo (força a prova a entrar em uma fase positiva, independente da anterior). Isso é feito acrescentando-se $⅋\bot$ ou $\otimes 1$ à fórmula com a qual se está trabalhando. É importante observar que as seguintes relações são válidas na lógica linear:

$$A \equiv A \otimes 1$$

$$A \equiv A ⅋ \bot$$

Porém, apesar da equivalência semântica, a prova *focused* das duas fórmulas apresenta comportamento diferente, como será mostrado a seguir.

Delay positivo

Seja F uma sub-fórmula de uma fórmula negativa na qual a prova está focada. Isso significa que a prova está em uma fase assíncrona. A decomposição dessa fórmula resultará, em algum momento, em F. Se F for positivo, uma regra do tipo [R⇑] será utilizada para colocá-la de volta ao contexto e ela só será utilizada novamente quando não houverem mais fórmulas negativas. Entretanto, se F for negativa, pela disciplina de provas utilizada, ela seria a próxima fórmula principal. Porém, isso poderia resultar em uma fase assíncrona muito extensa, e pode ser desejável que a prova de F seja separada da prova da fórmula que lhe deu origem por questões semânticas. Para isso é utilizado o *delay* positivo. Observe a prova abaixo:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\mathcal{K}_1 : \Gamma \Downarrow F} \quad \dfrac{}{\mathcal{K}_2 : \cdot \Downarrow 1}\,[1]}{\mathcal{K} : \Gamma \Downarrow F \otimes 1}\,[\otimes]}{\mathcal{K} : \Gamma, F \otimes 1 \Uparrow \cdot}\,[D_1]}{\mathcal{K} : \Gamma \Uparrow F \otimes 1}\,[\mathrm{R}\Uparrow]$$
$$\vdots$$

Note que na aplicação da regra $[D_1]$ poderia ser escolhida outra fórmula para a continuação da prova, o que desvincula o bloco assíncrono anterior da prova de F. Isso pode ser feito porque a utilização do *delay* positivo ($\otimes 1$) forçou F a ser colocada de volta ao contexto, e dessa maneira sua avaliação pôde ser atrasada.

Delay negativo

O *delay* negativo é análogo ao positivo. Seja F uma sub-fórmula de uma fórmula positiva na qual a prova está focada. Isso significa que a prova está em uma fase síncrona. A decomposição dessa fórmula resultará em F em algum momento. Se F for negativa, a prova continuará com essa fórmula, mudando para uma fase assíncrona. Entretanto, se F for positiva, pela disciplina seguida, a prova irá continuar com essa fórmula em uma fase síncrona. Da mesma forma que ocorre com a fase assíncrona, a existência de uma fase síncrona muito extensa pode ser indesejável, e para forçar a divisão da prova de F da sua superfórmula, pode ser usado um *delay* negativo.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{K} : \Gamma \Downarrow F}}{\mathcal{K} : \Gamma, F \Uparrow \cdot}\,[D_1]}{\mathcal{K} : \Gamma \Uparrow F}\,[\mathrm{R}\Uparrow]}{\mathcal{K} : \Gamma \Uparrow F, \bot}\,[\bot]}{\mathcal{K} : \Gamma \Uparrow F \parr \bot}\,[\parr]}{\mathcal{K} : \Gamma \Downarrow F \parr \bot}\,[\mathrm{R}\Downarrow]}{\vdots}$$

Observe que na aplicação da regra $[D_1]$ poderia ser escolhida outra fórmula, além de F, para a continuação da prova. Esse fato desvincula a prova da fórmula que deu origem a F da prova de F de fato. Isso foi possível devido a utilização do *delay* negativo ($\parr\bot$) que, iniciando uma fase assíncrona, permitiu com que F fosse colocado de volta no contexto e que sua avaliação fosse atrasada.

FASE ASSÍNCRONA

$$\frac{\vdash \mathcal{K} : \Gamma \Uparrow L, A \quad \vdash \mathcal{K} : \Gamma \Uparrow L, B}{\vdash \mathcal{K} : \Gamma \Uparrow L, A \& B}\ [\&] \qquad \frac{\vdash \mathcal{K} : \Gamma \Uparrow L, A, B}{\vdash \mathcal{K} : \Gamma \Uparrow L, A \parr B}\ [\parr]$$

$$\frac{}{\vdash \mathcal{K} : \Gamma \Uparrow L, \top}\ [\top] \qquad \frac{\vdash \mathcal{K} : \Gamma \Uparrow L}{\vdash \mathcal{K} : \Gamma \Uparrow L, \bot}\ [\bot]$$

$$\frac{\vdash \mathcal{K} : \Gamma \Uparrow L, A\{c/x\}}{\vdash \mathcal{K} : \Gamma \Uparrow L, \forall x.A}\ [\forall] \qquad \frac{\vdash \mathcal{K} +_l A : \Gamma \Uparrow L}{\vdash \mathcal{K} : \Gamma \Uparrow L, ?^l A}\ [?^l]$$

FASE SÍNCRONA

$$\frac{\vdash \mathcal{K} : \Gamma \Downarrow A_i}{\vdash \mathcal{K} : \Gamma \Downarrow A_1 \oplus A_2}\ [\oplus_i] \qquad \frac{\vdash \mathcal{K}_1 : \Gamma \Downarrow A \quad \vdash \mathcal{K}_2 : \Delta \Downarrow B}{\vdash \mathcal{K}_1 \otimes \mathcal{K}_2 : \Gamma, \Delta \Downarrow A \otimes B}\ [\otimes, \text{ dado que } (\mathcal{K}_1 = \mathcal{K}_2)|_{\mathcal{C} \cap \mathcal{W}}]$$

$$\frac{}{\vdash \mathcal{K} : \cdot \Downarrow 1}\ [1, \text{ dado que } \mathcal{K}[\mathcal{I} \setminus \mathcal{W}] = \emptyset] \qquad \frac{\vdash \mathcal{K} : \Gamma \Downarrow A\{t/x\}}{\vdash \mathcal{K} : \Gamma \Downarrow \exists x.A}\ [\exists]$$

$$\frac{\vdash \mathcal{K} \leq_l : \cdot \Uparrow A}{\vdash \mathcal{K} : \cdot \Downarrow !^l A}\ [!^l, \text{ dado que } \mathcal{K}[\{x \mid l \not\leq x \wedge x \notin \mathcal{W}\}] = \emptyset]$$

REGRAS ESTRUTURAIS E IDENTIDADE

$$\frac{}{\vdash \mathcal{K} : \Gamma \Downarrow A_p}\ [\text{I, dado que } A_p^\perp \in (\Gamma \cup \mathcal{K}[\mathcal{I}]) \text{ e } (\Gamma \cup \mathcal{K}[\mathcal{I} \setminus \mathcal{W}]) \subseteq \{A_p^\perp\}]$$

$$\frac{\vdash \mathcal{K} +_l P : \Gamma \Downarrow P}{\vdash \mathcal{K} +_l P : \Gamma \Uparrow \cdot}\ [D_l, \text{ dado que } l \in \mathcal{C} \cap \mathcal{W}] \qquad \frac{\vdash \mathcal{K} : \Gamma \Downarrow P}{\vdash \mathcal{K} +_l P : \Gamma \Uparrow \cdot}\ [D_l, \text{ dado que } l \notin \mathcal{C} \cap \mathcal{W}]$$

$$\frac{\vdash \mathcal{K} : \Gamma \Downarrow P}{\vdash \mathcal{K} : \Gamma, P \Uparrow \cdot}\ [D_1] \qquad \frac{\vdash \mathcal{K} : \Gamma \Uparrow N}{\vdash \mathcal{K} : \Gamma \Downarrow N}\ [R \Downarrow] \qquad \frac{\vdash \mathcal{K} : \Gamma, S \Uparrow L}{\vdash \mathcal{K} : \Gamma \Uparrow L, S}\ [R \Uparrow]$$

$$\frac{\mathcal{L} \vdash \mathcal{K}, C[s/loc, \hat{s}/\hat{loc}]}{\mathcal{L} \vdash \mathcal{K}, \Cup_l loc.C}\ [\Cup_l, \text{dado que } s \in \mathcal{L}]$$

$$\frac{\mathcal{L} \cup loc \vdash \mathcal{K}, C}{\mathcal{L} \vdash \mathcal{K}, \Cap_l loc.C}\ [\Cap_l, \text{dado que } loc \text{ é um novo location}]$$

Figura 4.7. Regras de inferência para a lógica linear SELLF com *focusing*. Nessa figura, a assinatura dos subexponenciais tem a seguinte propriedade: $\mathcal{C} \subseteq \mathcal{W}$. Além disso, L é uma lista de fórmulas, Γ é um multiconjunto de fórmulas e literais positivas, A_p é uma literal de polaridade positiva, P é uma literal não negativa, S é uma literal ou fórmula positiva e N é uma fórmula negativa.

Capítulo 5

Implementações

5.1 Implementação da lógica linear com subexponenciais

O sistema de provas SELLF da Figura 4.7 foi implementado na linguagem de programação lógica λ-Prolog [Reis & Pimentel, 2010]. Essa linguagem é a implementação de um fragmento da lógica clássica intuicionista (fórmulas de Harrop). Portanto, nesse caso, esse fragmento é a meta-lógica e a lógica linear com subexponenciais é a lógica-objeto. A maioria dos conectivos de SELLF podem ser especificados diretamente com os conectivos da lógica clássica intuicionista, já que a semântica difere apenas na manipulação do contexto. Observe por exemplo as regras para a conjunção em SELLF ($\otimes$ e $\&$). A conjunção aditiva ($\&$) tem a mesma semântica que a conjunção ($\wedge$) na lógica clássica (Figura 5.1). A diferença entre as versões aditiva e multiplicativa está no contexto das premissas e da conclusão.

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \ \wedge \ B} \ [\wedge]$$

Figura 5.1. Regra de conjunção à direita para a lógica clássica.

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \ \supset \ B} \ [\supset]$$

Figura 5.2. Regra de implicação à direita para a lógica clássica.

As fórmulas de Harrop, fragmento da lógica clássica intuicionista implementado como a linguagem λ-Prolog, não possuem qualquer controle sobre o contexto dos sequentes. A única operação permitida é a adição de fórmulas, que ocorre quando se utiliza uma implicação do lado direito do sequente (Figura 5.2). Logo, o contexto da meta-lógica é único e crescente.

Por ser uma lógica de recursos finitos, a lógica linear permite que fórmulas sejam consumidas, ou seja, "retiradas" do contexto quando utilizadas. Nessa lógica as fórmulas podem também ser acrescentadas ao contexto. Além disso, ainda é possível verificar se o contexto, ou parte dele, está vazio ou não, utilizando o operador $!^l$, como foi mostrado na Seção 4.2.1.

A presença dos subexponenciais oferece um controle maior sobre o contexto da lógica objeto, que pode agora ser interpretado como uma divisão em vários subcontextos, cada um correspondendo a um subexponencial e outro para a ausência dos mesmos. Cada um dos contextos $\mathcal{K}_i$ contém todas as fórmulas cujo operador principal seja $?^i$, e o contexto Γ contém as fórmulas que não possuem subexponencial $?^i$ algum. Devido a essa divisão, e à relação de pré-ordem entre esses índices, é possível agora inserir fórmulas em um subconjunto específico do contexto e conferir se um ou mais deles está vazio (Seção 4.1.2).

É fácil ver que a lógica objeto possui um controle muito maior do contexto que a meta lógica. Devido a essas diferenças, não é possível utilizar o contexto de λ-Prolog como o contexto de SELLF. Portanto, para a implementação da lógica-objeto o contexto foi representado explicitamente, como um dos argumentos do predicado principal. A estrutura escolhida foi uma lista de pares, em que cada par é formado por um identificador i, representando o índice de um subexponencial, e uma lista contendo as fórmulas que possuem o subexponencial $?^i$ como operador mais externo (sem o operador, i.e., uma fórmula F em $\mathcal{K}_i$ representa a fórmula $?^i F$ em um sequente que não possui essa divisão dos contextos).

Para que fosse possível implementar as regras de inferência da Figura 4.7, foi necessário implementar todas as operações realizadas sobre o contexto, mostradas na Figura 4.4. Elas são:

- $\mathcal{K} \leq_l$: retorna as fórmulas com subexponencial $?^i$ tal que $i \leq l$.
- $\mathcal{K} +_l A$: insere A no sub-contexto cujo identificador é l (que contém as fórmulas com marcador $?^l$). Essa função é usada na aplicação da regra $[?^l]$.
- $\mathcal{K}[S]$: retorna as fórmulas com subexponencial $?^i$ tal que $i \in S$.
- $\mathcal{K} = \mathcal{K}_1 \otimes \mathcal{K}_2$: divide o contexto respeitando a "linearidade" das fórmulas, i.e., se uma fórmula pode sofrer *contraction*, ela será colocada em $\mathcal{K}_1$ e $\mathcal{K}_2$ simultaneamente, como se tivesse sido duplicada.

Essas são operações relativamente simples de recursão em listas e portanto implementadas com facilidade em uma linguagem de programação lógica como λ-Prolog. Dadas essas operações, a implementação das regras de inferência de *SELLF* é imediata, já que são utilizados os conectivos da lógica intuicionista. Suponha a existência de um predicado `sellf`, que recebe como parâmetros o contexto e a fórmula focada. A implementação das regras de conjunção aditiva e multiplicativa é mostrada na Figura 5.3.

$$\text{sellf}(K, [(A \;\&\; B)|L]) \;\supset\; \text{sellf}(K, [A|L]) \;\wedge\; \text{sellf}(K, [B|L])$$
$$\text{sellf}(K, (A \otimes B)) \;\supset\; \text{split}(K, K_1, K_2) \;\wedge\; \text{sellf}(K_1, A) \;\wedge\; \text{sellf}(K_2, B)$$

Figura 5.3. Implementação das conjunções de *SELLF* em λ-Prolog.

Os operadores da lógica linear que não possuem correspondente na lógica clássica são os exponenciais (e subexponenciais), logo, a implementação desses operadores deve ser feita utilizando as fórmulas de Harrop de modo que sua semântica continue a mesma. Como as regras de inferência para esses operadores lidam principalmente com operações sobre o contexto, as implementações dessas operações são utilizadas. Para a regra de $?^i$, a fórmula é retirada do contexto e inserida no conjunto do subexponencial com índice i. Isso é feito

utilizando a operação $\mathcal{K}+_i A$, e retirando a fórmula $?^i A$ da lista onde estão as fórmulas focadas. Para a regra do $!^i$, é necessário conferir a condição com a operação $\mathcal{K}[S]$ e apagar as fórmulas necessárias (caso haja) com a operação $\mathcal{K} \leq_i$. A fórmula focada $!^i A$ continua no foco, mas sem o operador $!^i$. A implementação de ambas as regras estão mostradas na Figura 5.4.

$$\begin{aligned} \text{sellf}(K, [?^i A|L]) &\supset \text{sellf}(\mathcal{K} +_i A, L) \\ \text{sellf}(K, !^i A) &\supset \mathit{vazio}(\mathcal{K}[\{x \mid i \not\leq x \wedge x \notin \mathcal{W}\}]) \wedge \text{sellf}(\mathcal{K} \leq_i, A) \end{aligned}$$

Figura 5.4. Implementação dos subexponenciais de *SELLF* em λ-Prolog.

É importante observar que quando a prova está em uma fase assíncrona, o segundo argumento do predicado `sellf` é uma lista de fórmulas em vez de apenas uma. As regras D, que alteram a fase da prova de assíncrona para síncrona, consistem na escolha não-determinística de uma fórmula de K.

O código fonte dessa implementação está no Apêndice A.1.

5.2 Utilizando SELLF como uma meta-lógica

Utilizando a implementação de *SELLF* descrita na Seção 5.1 foi possível implementar a especificação em lógica linear com subexponenciais de alguns sistemas de prova (Seção 5.2.1) e de uma linguagem de programação simples (Seção 5.2.2).

Dado que *SELLF* foi implementado em λ-Prolog, pode-se assumir a existência de um predicado `sellf`, que recebe como argumentos um sequente da lógica linear focada (contexto clássico[1], contexto linear e objetivo) e tenta provar o objetivo utilizando as fórmulas dos contextos. Essa implementação possibilita o uso de *SELLF* como uma meta-lógica.

Um sistema especificado em *SELLF* consiste de uma série de fórmulas na lógica linear com subexponenciais que descrevem a semântica de cada comando do sistema. A prova de uma fórmula deve corresponder à execução do comando que ela representa. No caso da especificação de sistemas de cálculo de sequente de outras lógicas, cada fórmula corresponde à definição da regra de introdução um conectivo lógico ou uma regra estrutural (*weakening*, *contraction*, etc.). No caso da linguagem de programação, cada fórmula corresponde a um construto da linguagem (*load*, *unload*, etc.). Dado um sequente na lógica-objeto ou um programa na linguagem especificada, a prova do sequente ou execução do programa podem ser obtidos fazendo a tradução (*parsing*) dos comandos ou fórmulas e usando o predicado `sellf` para provar as fórmulas em LL. Essa prova em lógica linear corresponderá à prova do sequente na lógica objeto ou execução dos comandos na linguagem de programação.

Entretanto, a presença de várias fórmulas aninhadas e comandos com definições recursivas torna difícil que a tradução seja feita em um só passo. A resolução desse problema é a avaliação preguiçosa (*lazy evaluation*) do objetivo. O conceito de *definitions* permite que a tradução seja feita pelo próprio interpretador, apenas quando necessário, com a condição que a especificação do sistema completo esteja no contexto clássico.

Dada uma fórmula a na lógica objeto[2] e sua especificação B em *SELLF* (meta-lógica), a expressão:

[1] devidamente dividido pelos subexponenciais

[2] a também poderia ser um comando do sistema especificado, porém as denominações lógica objeto e fórmula serão utilizadas no resto dessa seção.

$$a \triangleq B$$

representa sua definição (*definition*). Ela é equivalente à seguinte implicação em LL:

$$B \multimap a \equiv B^{\perp} \parr a$$

que por sua vez é equivalente à fórmula da direita. Essa implicação indica que a pode ser reescrito com B. A prova da Figura 5.5 mostra a sequência de aplicações de regras que resulta na reescrita. Note que a fórmula $B^{\perp} \parr a$ está negada porque o sistema utilizado é *one-sided*, e que a reescrita é feita ao focar nela. A fórmula a da lógica objeto é interpretada como um átomo na meta-lógica, e isso permite que a sub-árvore de prova da direita termine, retirando a do contexto linear, e que a prova continue com B, que é de fato a especificação de a, e o mesmo contexto. Observe que a fórmula $B \otimes a^{\perp}$ (definição de a negada) deve estar em um contexto clássico, e não é consumida após a aplicação da regra D_1.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\vdash \mathcal{K} : \Gamma \Downarrow B} \quad \dfrac{}{\vdash \cdot : a \Downarrow a^{\perp}}\,[\mathrm{I}]}{\vdash \mathcal{K} : \Gamma, a \Downarrow B \otimes a^{\perp}}\,[\otimes]}{\vdash \mathcal{K} : \Gamma, a \Uparrow \cdot}\,[D_1] \text{ dado que } (B \otimes a^{\perp}) \in \mathcal{K}}{\dfrac{\vdots}{\vdash \mathcal{K} : \Gamma, a \Uparrow L}}}{\vdash \mathcal{K} : \Gamma \Uparrow a, L}\,[\mathrm{R}\Uparrow]$$

Figura 5.5. Reescrita de a como B.

Para cada definição $a \triangleq B$ de uma especificação, a fórmula $B \otimes a^{\perp}$ é colocada no contexto clássico, i.e., em um subexponencial que pode sofrer *contraction* e *weakening*, já que pode ser utilizada sempre que for necessário obter a definição de uma fórmula da lógica objeto. O predicado `sellf` é então chamado com essas fórmulas no contexto, e seu objetivo será o sequente da lógica objeto que se deseja provar (ou o programa que se deseja executar).

5.2.1 Especificação de sistemas de cálculo de sequentes

A principal vantagem da lógica linear com subexponenciais sobre a lógica linear é o aumento da expressividade, i.e., com a primeira é possível especificar mais sistemas do que com a segunda. Na lógica linear tradicional, é possível especificar sistemas de sequentes cuja estrutura possua apenas um conjunto (contexto clássico) e um multiconjunto (contexto linear) de fórmulas. A existência de subexponenciais permite que se tenha quantos conjuntos ou multiconjuntos se queira. Nessa seção são apresentados alguns sistemas de cálculo de sequentes cujas restrições estruturais tornam sua especificação complicada para a lógica linear, mas feita diretamente em *SELLF* [Nigam et al., 2010].

Todos os sistemas foram implementados utilizando o interpretador de *SELLF* em λ-Prolog já descrito na Seção 5.1. As características em comum dessas implementações são descritas a seguir. Nas seções correspondentes a cada sistema estão explicadas as particularidades de cada um.

Para permitir a interpretação e tradução das fórmulas da lógica objeto para a meta-lógica, foi utilizado o método das definições de comandos descrito na Seção 5.2. Essas

definições (fórmulas do tipo $B \otimes a^{\perp}$) são armazenadas em um subexponencial especial, denotado ∞, cujas fórmulas podem sofrer *contraction* e *weakening*.

Foi definido um tipo $form$ para as fórmulas da lógica objeto, assim como seus conectivos e os tipos de cada um deles. O mapeamento de fórmulas da lógica objeto para átomos da meta-lógica é feito por meio dos predicados $\lceil\cdot\rceil$ e $\lfloor\cdot\rfloor$, que têm tipo $form \rightarrow o$ (o é o tipo das fórmulas da meta-lógica). $\lceil F\rceil$ é utilizado quando F ocorre do lado direito do sequente e $\lfloor F\rfloor$ é utilizado quando ela ocorre do lado esquerdo. Essa diferenciação é importante, já que as definições para conectivos que ocorrem do lado esquerdo ou direito de um sequente são distintas. Dessa maneira, é possível substituir a fórmula por sua definição correta.

Para todos os sistemas foi implementado um predicado principal que recebe como parâmetros a estrutura do sequente que se quer provar na lógica objeto. Por exemplo, se o sequente da lógica objeto é formado por algumas fórmulas do lado esquerdo e uma fórmula do lado direito (como o da lógica intuicionista), o predicado principal receberia dois parâmetros: uma lista de fórmulas que ocorrem do lado esquerdo e a fórmula que ocorre do lado direito. O interpretador é responsável por "atomizar" essas fórmulas, aplicando a elas os predicados $\lceil\cdot\rceil$ ou $\lfloor\cdot\rfloor$. Estes átomos são colocados em uma lista que será o objetivo do sequente em SELLF. As definições das fórmulas da lógica objeto são colocadas no subexponencial ∞ e todos os outros índices subexponenciais necessários são declarados com nenhuma fórmula. Após a construção desse sequente, o interpretador de *SELLF* é chamado, e ele conseguirá provar esse sequente se, e somente se, o sequente da lógica objeto também for provável. Essa correspondência foi provada por Vivek Nigam em [Nigam, 2009] para os sistemas descritos a seguir.

5.2.1.1 Implementação da lógica *G1m*

A Figura 5.6 mostra as regras de inferência do sistema de cálculo de sequentes para a lógica minimal, chamado *G1m* [Troelstra & Schwichtenberg, 1996]. Nesse sistema, as regras de *contraction* e *weakening* são explícitas, e portanto as fórmulas do lado esquerdo e direito do sequente serão representadas por multiconjuntos na lógica linear. Para especificar o sistema *G1m*, serão utilizados dois subexponenciais, l para as fórmulas da esquerda e r para as fórmulas da direita (do inglês *left* e *right*), cujos elementos não poderão sofrer *contraction* ou *weakening*. A aplicação de uma dessas regras estruturais em uma fórmula na lógica objeto será representada pela prova em lógica linear da fórmula que especifica a regra, ao invés da aplicação direta de *contraction* ou *weakening* na meta-lógica. Essa é uma diferença importante para separar uma prova na meta-lógica de uma prova na lógica objeto.

Os índices subexponenciais dessa especificação estão relacionados pela seguinte ordenação parcial: $l \preceq \infty$ e $r \preceq \infty$. Os operadores $?^l$ e $?^r$ colocam as fórmulas da lógica objeto (átomos na meta-lógica) no subexponencial correto, dependendo se são fórmulas que ocorrem do lado direito ou esquerdo do sequente. O operador $!^l$ é utilizado nas regras $\supset L$ e Cut para garantir que a fórmula C seja passada para o sequente da direita. Dessa maneira, a fórmula A (criada pelo Cut ou antecedente na fórmula $A \supset B$) colocada do lado direito do sequente é única, assim como na especificação do sistema. Esse processo é ilustrado pela seguinte derivação da especificação da regra Cut:

$$\dfrac{\dfrac{\vdash \mathcal{L}_{G1m} \dot{\circ} \lfloor \Gamma_1 \rfloor \,\dot{i}\, \lceil A \rceil \,\dot{r}\cdot \Uparrow}{\vdash \mathcal{L}_{G1m} \dot{\circ} \lfloor \Gamma_1 \rfloor \,\dot{i}\cdot \dot{r}\cdot \Downarrow !^l ?^r \lceil A \rceil}\, !^l, ?^r \quad \dfrac{\vdash \mathcal{L}_{G1m} \dot{\circ} \lfloor \Gamma_2, A \rfloor \,\dot{i}\, \lceil C \rceil \,\dot{r}\cdot \Uparrow}{\vdash \mathcal{L}_{G1m} \dot{\circ} \lfloor \Gamma_2 \rfloor \,\dot{i}\, \lceil C \rceil \,\dot{r}\cdot \Downarrow ?^l \lfloor A \rfloor}\, R\Downarrow, ?^l}{\dfrac{\vdash \mathcal{L}_{G1m} \dot{\circ} \lfloor \Gamma_1, \Gamma_2 \rfloor \,\dot{i}\, \lceil C \rceil \,\dot{r}\cdot \Downarrow !^l ?^r \lceil A \rceil \otimes ?^l \lfloor A \rfloor}{\vdash \mathcal{L}_{G1m} \dot{\circ} \lfloor \Gamma_1, \Gamma_2 \rfloor \,\dot{i}\, \lceil C \rceil \,\dot{r}\cdot \Uparrow}\, D_\infty, \exists}\, \otimes$$

Quando o contexto é dividido na aplicação da regra $\otimes$, o átomo $\lceil C \rceil$ deve ser colocado no sequente à direita. Caso contrário, não seria possível aplicar a regra $!^l$.

O código fonte dessa implementação está no Apêndice A.2.

$$\dfrac{\Gamma_1 \longrightarrow A \quad \Gamma_2, B \longrightarrow C}{\Gamma_1, \Gamma_2, A \supset B \longrightarrow C}\, \supset L \qquad \dfrac{\Gamma, A \longrightarrow B}{\Gamma \longrightarrow A \supset B}\, \supset R \qquad \dfrac{\Gamma, A_i \longrightarrow C}{\Gamma, A_1 \wedge A_2 \longrightarrow C}\, \wedge_i L$$

$$\dfrac{\Gamma_1 \longrightarrow A \quad \Gamma_2 \longrightarrow B}{\Gamma_1, \Gamma_2 \longrightarrow A \wedge B}\, \wedge R \qquad \dfrac{\Gamma, A\{t/x\} \longrightarrow C}{\Gamma, \forall x\, A \longrightarrow C}\, \forall L \qquad \dfrac{\Gamma \longrightarrow A\{c/x\}}{\Gamma \longrightarrow \forall x\, A}\, \forall R$$

$$\dfrac{\Gamma, A\{c/x\} \longrightarrow C}{\Gamma, \exists x\, A \longrightarrow C}\, \exists L \qquad \dfrac{\Gamma \longrightarrow A\{t/x\}}{\Gamma \longrightarrow \exists x\, A}\, \exists R \qquad \dfrac{\Gamma, A \longrightarrow C \quad \Gamma, B \longrightarrow C}{\Gamma, A \vee B \longrightarrow C}\, \vee L$$

$$\dfrac{\Gamma \longrightarrow A_i}{\Gamma \longrightarrow A_1 \vee A_2}\, \vee_i R \qquad \dfrac{\Gamma \longrightarrow C}{\Gamma, A \longrightarrow C}\, W_L \qquad \dfrac{\Gamma, A, A \longrightarrow C}{\Gamma, A \longrightarrow C}\, C_L$$

$$\dfrac{}{A \longrightarrow A}\, \mathrm{I} \qquad \dfrac{\Gamma_1 \longrightarrow A \quad \Gamma_2, A \longrightarrow C}{\Gamma_1, \Gamma_2 \longrightarrow C}\, \mathrm{Cut}$$

Figura 5.6. Cálculo de sequentes *G1m* para a lógica minimal. Nessa figura, Γ_1 e Γ_2 são multiconjuntos de fórmulas e C é uma fórmula; nas regras $\exists L$ e $\forall R$, a variável c não é livre em Γ ou C; e $i \in \{1, 2\}$.

$(\supset_L)$	$\lfloor A \supset B \rfloor^\perp \otimes (!^l ?^r \lceil A \rceil \otimes ?^l \lfloor B \rfloor)$	$(\supset_R)$	$\lceil A \supset B \rceil^\perp \otimes (?^l \lfloor A \rfloor \parr ?^r \lceil B \rceil)$
$(\wedge_L)$	$\lfloor A \wedge B \rfloor^\perp \otimes (?^l \lfloor A \rfloor \oplus ?^l \lfloor B \rfloor)$	$(\wedge_R)$	$\lceil A \wedge B \rceil^\perp \otimes (?^r \lceil A \rceil \otimes ?^r \lceil B \rceil)$
$(\vee_L)$	$\lfloor A \vee B \rfloor^\perp \otimes (?^l \lfloor A \rfloor \,\&\, ?^l \lfloor B \rfloor)$	$(\vee_R)$	$\lceil A \vee B \rceil^\perp \otimes (?^r \lceil A \rceil \oplus ?^r \lceil B \rceil)$
$(\forall_L)$	$\lfloor \forall B \rfloor^\perp \otimes ?^l \lfloor Bx \rfloor$	$(\forall_R)$	$\lceil \forall B \rceil^\perp \otimes \forall x ?^r \lceil Bx \rceil$
$(\exists_L)$	$\lfloor \exists B \rfloor^\perp \otimes \forall x ?^l \lfloor Bx \rfloor$	$(\exists_R)$	$\lceil \exists B \rceil^\perp \otimes ?^r \lceil Bx \rceil$
(I)	$\lfloor B \rfloor^\perp \otimes \lceil B \rceil^\perp$	(Cut)	$!^l ?^r \lceil B \rceil \otimes ?^l \lfloor B \rfloor$
(C_L)	$\lfloor B \rfloor^\perp \otimes (?^l \lfloor B \rfloor \parr ?^l \lfloor B \rfloor)$	(W_L)	$\lfloor B \rfloor^\perp \otimes \perp$

Figura 5.7. $\mathcal{L}_{G1m}$: Especificação para o sistema *G1m*.

5.2.1.2 Implementação da lógica *mLJ*

O sistema *mLJ*, descrito na Figura 5.8, é um sistema de provas para a lógica intuicionista de multi-conclusão [Maehara, 1954]. A diferença principal para o sistema da lógica intuicionista tradicional (*LJ*) é a permissão da existência de mais de uma fórmula do lado direito do sequente. A regra inicial permite que existam fórmulas do lado direito e esquerdo do sequente além da principal (A) e as regras de *contraction* e *weakening* não estão explícitas (note que

fórmulas são "apagadas" do contexto quando se aplicam as regras de $\supset_r$ ou $\forall_r$), logo, na especificação em *SELLF*, os sequentes em *mLJ* possuem dois conjuntos de fórmulas (dois contextos clássicos diferentes), um para cada lado direito e esquerdo.

Serão utilizados, para isso, dois índices de subexponenciais, e ambos devem permitir as operações de *contraction* e *weakening* para que sejam conjuntos. Estes subexponenciais foram chamados novamente de l e r. As relações de pré-ordem existentes, incluindo o ∞, são: $l \preceq \infty, r \preceq \infty$.

A especificação de *mLJ* é mostrada na figura 5.9. Os operadores subexponenciais $?^l$ e $?^r$ são utilizados para inserir as fórmulas nos contextos l e r respectivamente. O operador $!^l$ é utilizado na especificação das regras de inferência $\supset_r$ e $\forall_r$, para garantir que as fórmulas do lado direito do sequente sejam apagadas como necessário. A derivação abaixo mostra a prova da fórmula que especifica a regra $\forall_r$:

$$\dfrac{\dfrac{}{\vdash \mathcal{L}_{MLJ} \dot{\infty} \lfloor \Gamma \rfloor \dot{l} \lceil \Delta, \forall x\, A \rceil \dot{r} \cdot \Downarrow \lceil \forall x\, A \rceil^{\perp}}\, I_r \quad \dfrac{\dfrac{\vdash \mathcal{L}_{MLJ} \dot{\infty} \lfloor \Gamma \rfloor \dot{l} \lceil Ac \rceil \dot{r} \cdot \Uparrow}{\vdash \mathcal{L}_{MLJ} \dot{\infty} \lfloor \Gamma \rfloor \dot{l} \cdot \dot{r} \cdot \Uparrow \forall x ?^r \lceil Ax \rceil}\, \forall, ?^r}{\vdash \mathcal{L}_{MLJ} \dot{\infty} \lfloor \Gamma \rfloor \dot{l} \lceil \Delta, \forall x\, A \rceil \dot{r} \cdot \Downarrow !^l \forall x ?^r \lceil Ax \rceil}\, !^l}{\dfrac{\vdash \mathcal{L}_{MLJ} \dot{\infty} \lfloor \Gamma \rfloor \dot{l} \lceil \Delta, \forall x\, A \rceil \dot{r} \cdot \Downarrow \lceil \forall A \rceil^{\perp} \otimes !^l \forall x ?^r \lceil Ax \rceil}{\vdash \mathcal{L}_{MLJ} \dot{\infty} \lfloor \Gamma \rfloor \dot{l} \lceil \Delta, \forall x\, A \rceil \dot{r} \cdot \Uparrow}\, D_\infty, \exists}\, \otimes$$

O código fonte dessa implementação está no Apêndice A.3.

5.2.1.3 Implementação da lógica *LJQ**

Os sistemas apresentados anteriormente podem ser especificados com apenas dois subexponenciais, que representam conjuntos ou multiconjuntos. Isso se deve à estrutura dos sequentes. Entretanto, existem sistemas que necessitam de mais de dois contextos para que sejam especificados, por exemplo aqueles que utilizam *focusing*, um deles é o *LJQ** [Dyckhoff & Lengrand, 2006], apresentado nessa seção.

A Figura 5.10 mostra o cálculo de sequentes para *LJQ**, um sistema focado para a lógica intuicionista com multi-conclusão. Como se pode perceber, seus sequentes possuem dois formatos: $\Gamma \vdash \Delta$ (não focados) e $\Gamma \vdash A; \Delta$ (com a fórmula A focada). As regras de inferência à esquerda são aplicadas apenas em sequentes não focados, e as regras à direita são aplicadas apenas em sequentes focados (na fórmula que está focada). Para especificar os sequentes focados, será necessário mais um subexponencial, além de l e r que foram usados até agora. Esse índice subexponencial será chamado de f e armazenará a fórmula focada, quando houver. Nessa especificação, l e r poderão sofrer *contraction* e *weakening* (são conjuntos), mas nenhuma dessas operações podem ser aplicadas para elementos de f (multiconjunto). A ordenação parcial desses subexponenciais é dada por: $l \preceq r \preceq \infty$. Nesse caso, l e r estão relacionados, diferente das especificações anteriores, e $r \not\preceq f$, o que faz com que, na aplicação da regra $\wedge L$, por exemplo, o contexto f esteja garantidamente vazio e l permaneça inalterado. A derivação abaixo corresponde à aplicação dessa regra, nela, Γ' é o conjunto $\Gamma \uplus \{A \wedge B\}$ e $\mathcal{K}$ é uma abreviação para $\mathcal{L}_{LJQ} \dot{\infty} \lfloor \Gamma' \rfloor \dot{l} \lceil \Delta \rceil \dot{r} \cdot \dot{f} \cdot$.

$$\frac{\Gamma, A \supset B \longrightarrow A, \Delta \quad \Gamma, A \supset B, B \longrightarrow \Delta}{\Gamma, A \supset B \longrightarrow \Delta} \supset_l \qquad \frac{\Gamma, A \longrightarrow B}{\Gamma \longrightarrow A \supset B, \Delta} \supset_r$$

$$\frac{\Gamma, A \wedge B, A, B \longrightarrow \Delta}{\Gamma, A \wedge B \longrightarrow \Delta} \wedge_l \qquad \frac{\Gamma \longrightarrow A \wedge B, A, \Delta \quad \Gamma \longrightarrow A \wedge B, B, \Delta}{\Gamma \longrightarrow A \wedge B, \Delta} \wedge_r$$

$$\frac{\Gamma, A \vee B, A, \longrightarrow \Delta \quad \Gamma, A \vee B, B \longrightarrow \Delta}{\Gamma, A \vee B \longrightarrow \Delta} \vee_l \qquad \frac{\Gamma \longrightarrow A \vee B, A, B, \Delta}{\Gamma \longrightarrow A \vee B, \Delta} \vee_r$$

$$\frac{\Gamma, \forall x\, A, A\{t/x\} \longrightarrow \Delta}{\Gamma, \forall x\, A \longrightarrow \Delta} \forall_l \qquad \frac{\Gamma \longrightarrow A\{c/x\}}{\Gamma \longrightarrow \Delta, \forall x\, A} \forall_r$$

$$\frac{\Gamma, \exists x\, A, A\{c/x\} \longrightarrow \Delta}{\Gamma, \exists x\, A \longrightarrow \Delta} \exists_l \qquad \frac{\Gamma \longrightarrow \Delta, \exists x\, A, A\{t/x\}}{\Gamma \longrightarrow \Delta, \exists x\, A} \exists_r$$

$$\frac{}{\Gamma, A \longrightarrow A, \Delta}\ \mathrm{I} \qquad \frac{\Gamma \longrightarrow B, \Delta \quad \Gamma, B \longrightarrow \Delta}{\Gamma \longrightarrow \Delta}\ \mathrm{Cut} \qquad \frac{}{\Gamma, \bot \longrightarrow \Delta}\ \bot_l$$

Figura 5.8. Cálculo de sequentes para a lógica intuicionista com multi-conclusão *mLJ*.

$(\supset_l)$	$\lfloor A \supset B \rfloor^\bot \otimes (?^r \lceil A \rceil \& ?^l \lfloor B \rfloor)$	$(\supset_r)$	$\lceil A \supset B \rceil^\bot \otimes !^l (?^l \lfloor A \rfloor \parr ?^r \lceil B \rceil)$
$(\wedge_l)$	$\lfloor A \wedge B \rfloor^\bot \otimes (?^l \lfloor A \rfloor \parr ?^l \lfloor B \rfloor)$	$(\wedge_r)$	$\lceil A \wedge B \rceil^\bot \otimes (?^r \lceil A \rceil \& ?^r \lceil B \rceil)$
$(\vee_l)$	$\lfloor A \vee B \rfloor^\bot \otimes (?^l \lfloor A \rfloor \& ?^l \lfloor B \rfloor)$	$(\vee_r)$	$\lceil A \vee B \rceil^\bot \otimes (?^r \lceil A \rceil \parr ?^r \lceil B \rceil)$
$(\forall_l)$	$\lfloor \forall B \rfloor^\bot \otimes ?^l \lfloor Bx \rfloor$	$(\forall_r)$	$\lceil \forall B \rceil^\bot \otimes !^l \forall x ?^r \lceil Bx \rceil$
$(\exists_l)$	$\lfloor \exists B \rfloor^\bot \otimes \forall x ?^l \lfloor Bx \rfloor$	$(\exists_r)$	$\lceil \exists B \rceil^\bot \otimes ?^r \lceil Bx \rceil$
$(\bot_l)$	$\lfloor \bot \rfloor^\bot$		
(I)	$\lfloor B \rfloor^\bot \otimes \lceil B \rceil^\bot$	(Cut)	$?^l \lfloor B \rfloor \otimes ?^r \lceil B \rceil$

Figura 5.9. $\mathcal{L}_{mLJ}$: Especificação para o sistema lógico intuicionista com multi-conclusão *mLJ*.

$$\dfrac{\dfrac{}{\vdash \mathcal{K} \Downarrow \lfloor A \wedge B \rfloor^\bot}\ I_l \qquad \dfrac{\vdash \mathcal{L}_{LJQ} \stackrel{\cdot}{i} \lfloor \Gamma', A, B \rfloor \stackrel{\cdot}{r} \lceil \Delta \rceil \stackrel{\cdot}{f} \cdot : \cdot \Uparrow}{\vdash \mathcal{L}_{LJQ} \stackrel{\cdot}{i} \lfloor \Gamma' \rfloor \stackrel{\cdot}{r} \lceil \Delta \rceil \stackrel{\cdot}{f} \cdot : \cdot \Downarrow !^r (?^l \lfloor A \rfloor \parr ?^l \lfloor B \rfloor)}\ !^r, \parr, 2\times ?^l}{\vdash \mathcal{L}_{LJQ} \stackrel{\cdot}{i} \lfloor \Gamma' \rfloor \stackrel{\cdot}{r} \lceil \Delta \rceil \stackrel{\cdot}{f} \cdot : \cdot \Downarrow \lfloor A \wedge B \rfloor^\bot \otimes !^r (?^l \lfloor A \rfloor \parr ?^l \lfloor B \rfloor)}\ \otimes}{\vdash \mathcal{L}_{LJQ} \stackrel{\cdot}{i} \lfloor \Gamma' \rfloor \stackrel{\cdot}{r} \lceil \Delta \rceil \stackrel{\cdot}{f} \cdot : \cdot \Uparrow}\ D_\infty, 2 \times \exists$$

O código fonte dessa implementação está no Apêndice A.4.

5.2.2 Implementação da linguagem BAG

Recentemente, Nigam [Nigam, 2009] apresentou a especificação da semântica de uma linguagem de programação simples utilizando a lógica linear com subexponenciais. Essa linguagem, apesar da sua simplicidade, é Turing-completa, i.e., consegue simular uma máquina de Turing, pois possui instruções de leitura e escrita em memória, condicionais e repetição. Isso significa que toda execução de um algoritmo em uma máquina de Turing pode ser represen-

$$\frac{\Gamma, A \supset B \to A; \cdot \quad \Gamma, A \supset B, B \vdash \Delta}{\Gamma, A \supset B \vdash \Delta}\ \supset_l \qquad \frac{\Gamma, A \vdash B}{\Gamma \to A \supset B; \Delta}\ \supset_r$$

$$\frac{\Gamma, A \vee B, A \vdash \Delta \quad \Gamma, A \vee B, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}\ \vee_l \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \to A \vee B; \Delta}\ \vee_r$$

$$\frac{\Gamma, A \wedge B, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\ \wedge_l \qquad \frac{\Gamma \to A; \Delta \quad \Gamma \to B; \Delta}{\Gamma \to A \wedge B; \Delta}\ \wedge_r$$

$$\frac{}{\Gamma, A \to A; \Delta}\ \mathrm{I} \qquad \frac{\Gamma \to C; \Delta}{\Gamma \vdash C, \Delta}\ D \qquad \frac{}{\Gamma, \bot \vdash \Delta}\ \bot_l$$

Figura 5.10. Cálculo de sequentes para o sistema focado com multi-conclusão para a lógica intuicionista LJQ.

(Id_1)	$\lfloor A \rfloor^\perp \otimes \lceil A \rceil^\perp$	$(\bot_L)$	$\lfloor \bot \rfloor^\perp$
$(\supset_L)$	$\lfloor A \supset B \rfloor^\perp \otimes (!^l ?^f \lceil A \rceil \otimes !^r ?^l \lfloor B \rfloor)$	$(\supset_R)$	$\lceil A \supset B \rceil^\perp \otimes !^l (?^l \lfloor A \rfloor \parr ?^r \lceil B \rceil)$
$(\vee_L)$	$\lfloor A \vee B \rfloor^\perp \otimes (!^r ?^l \lfloor A \rfloor \otimes !^r ?^l \lfloor B \rfloor)$	$(\vee_R)$	$\lceil A \vee B \rceil^\perp \otimes !^r (?^r \lceil A \rceil \parr ?^r \lceil B \rceil)$
$(\wedge_L)$	$\lfloor A \wedge B \rfloor^\perp \otimes !^r (?^l \lfloor A \rfloor \parr ?^l \lfloor B \rfloor)$	$(\wedge_R)$	$\lceil A \wedge B \rceil^\perp \otimes (!^r ?^f \lceil A \rceil \otimes !^r ?^f \lceil B \rceil)$

Figura 5.11. $\mathcal{L}_{LJQ*}$: Especificação do sistema LJQ^*.

tada por uma prova em *SELLF*. A Seção 5.3 apresenta a semântica de cada comando com sua respectiva árvore de prova. Com isso, prova-se que o conjunto de algoritmos está contido no conjunto de provas da lógica linear com subexponenciais ($algoritmos \subset provas\ em\ SELLF$).

A implementação de BAG segue a mesma linha da especificação de sistemas de cálculo de sequentes, com algumas leves alterações. O conceito de definições ainda é utilizado, e portanto, para cada comando c cuja semântica é descrita pela fórmula F temos a fórmula $c^\perp \otimes F$ em um contexto clássico chamado def. O tipo dos comandos na linguagem é $prog$, e o predicado `cmd`, cujo tipo é $prog \to o$, é responsável por mapear construtos de BAG para átomos de *SELLF*. O predicado principal, chamado `bag`, recebe como parâmetro um programa em BAG e constrói o sequente da lógica linear com subexponenciais para ser provado em *SELLF*. Esse sequente é formado pelo contexto clássico def com as definições dos comandos, além de outros que sejam necessários para a execução do programa, e possui como objetivo o programa atomizado pelo predicado `cmd`. O código correspondente a essa implementação está no Apêndice A.5.

Durante a implementação e testes, notou-se a necessidade de alteração da semântica de alguns comandos, de maneira que as fórmulas apresentadas nesse trabalho diferem ligeiramente daquelas apresentadas em [Nigam, 2009].

Toda a descrição da linguagem é feita separadamente, na Seção 5.3.

5.3 A linguagem BAG

A linguagem BAG é uma linguagem de programação que foi especificada utilizando a lógica linear. Isso significa que foi associado a cada comando uma fórmula da lógica linear, cuja prova representa a execução desse comando corretamente. O exemplo mais simples é o comando **end**, que é definido como $\top$ (verdadeiro). No sistema de provas focado da lógica linear, a existência de um $\top$ significa a terminação de uma prova. Isso corresponde ao final da

execução de um programa. Os outros comandos apresentam fórmulas mais complexas, mas todas correspondem à semântica desejada.

A especificação dessa linguagem mostra como algoritmos podem ser especificados utilizando lógica. Para cada programa (algoritmo ou sistema) implementado em BAG, existe uma fórmula da lógica linear correspondente. A execução desse programa é representada pela árvore de prova da fórmula correspondente utilizando o sistema de provas focado da lógica linear. Dessa maneira podemos mostrar que o conjunto de especificações em BAG está contido no conjunto de fórmulas da lógica linear.

5.3.1 Sintaxe

Os domínios sintáticos de BAG são os seguintes:

- $\mathbb{C}$: conjunto de constantes e outros *tokens* necessários.
- $\mathcal{V}$: variáveis que representam elementos de $\mathbb{C}$.
- $\mathcal{K}$: variáveis que representam continuação de programa (observe que esse conjunto é diferente do contexto $\mathcal{K}$ de *SELLF*).
- $\mathcal{L}$: conjunto de *locations*.
- $\mathcal{N}$: conjunto de nomes de módulos.

Assume-se que $var \in \mathcal{V}$, $K \in \mathcal{K}$, $L \in \mathcal{L}$ e $name \in \mathcal{N}$.
As classes sintáticas de BAG são definidas na Figura 5.12:

$$
\begin{aligned}
t &::= c \in \mathbb{C} \mid var \\
tup &::= \langle t_1, ..., t_n \rangle \ (n \geq 0) \\
pat &::= tup \mid \lambda var.pat \\
cond &::= cond_a \mid cond_l \\
cond_a &::= t_1 \blacktriangledown t_2 \ (\blacktriangledown \in \{\leq, <, =, >, \geq\}) \\
cond_l &::= \textbf{is\_empty}\ L \\
bprog &::= prog \mid \lambda var.bprog \\
kprog &::= \lambda K.prog \mid \lambda var.kprog \\
lprog &::= \lambda K.prog \mid \lambda L.lprog \mid \lambda var.lprog \\
prog &::= \textbf{load}\ tup\ loc\ prog \mid \textbf{unload}_i\ loc\ pat\ bprog \\
&\mid \textbf{while}\ cond_a\ (\lambda K.prog)\ prog \mid \textbf{loop}_i\ loc\ kprog\ prog \\
&\mid \textbf{new}\ loc\ \lambda L.prog \mid prog_1\ []\ prog_2 \\
&\mid \textbf{if}\ cond\ \textbf{then}\ kprog_1\ \textbf{else}\ kprog_2\ prog \\
&\mid K \mid plus\ x\ y\ z \mid mult\ x\ y\ z \mid \textbf{end}.
\end{aligned}
$$

Figura 5.12. Classes sintáticas de BAG.

5.3.2 Semântica

Nessa seção é definida a semântica de BAG utilizando a lógica linear. Será apresentada a definição formal, a árvore de prova correspondente à execução do comando e uma discussão sobre cada trecho.

As regras [def⇑] e [def⇓] são macros para o processo de substituição de um construto da linguagem pela sua definição na meta-lógica. Esse processo é descrito com detalhes na Seção 5.2.

5.3.2.1 Comando `load`

$$\mathbf{load}\ tup\ l\ prog\ \triangleq\ ?^l l(tup) \parr \delta^+(prog)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash \mathcal{K}_{+l} l(tup) : \cdot \Downarrow \delta^+(prog)}{\vdash \mathcal{K}_{+l} l(tup) : \delta^+(prog) \Uparrow \cdot}\,[D_1]}{\vdash \mathcal{K}_{+l} l(tup) : \cdot \Uparrow \delta^+(prog)}\,[\mathrm{R}\Uparrow]}{\vdash \mathcal{K} : \cdot \Uparrow ?^l l(tup), \delta^+(prog)}\,[?^l]}{\vdash \mathcal{K} : \cdot \Uparrow ?^l l(tup) \parr \delta^+(prog)}\,[\parr]}{\vdash \mathcal{K} : \cdot \Uparrow \mathbf{load}\ tup\ l\ prog}\,[\mathrm{def}\Uparrow]$$

O comando **load** coloca a tupla *tup* no location l utilizando a operação $\mathcal{K}_{+l}$. Essa sequência de passos é realizada sem *backtracking*, ou seja, não existe a possibilidade de falha. Note que a utilização do *delay* positivo na continuação (*prog*) permite que outra fórmula do contexto linear seja escolhida pela regra D_1 para ser executada, e também marca o final da execução do comando **load** e início de um outro comando do programa.

5.3.2.2 Comando `unload`

$$\mathbf{unload}_i\ l\ pat\ bprog\ \triangleq\ l(pat\ v_1, \cdots, v_i)^\perp\ \otimes\ \delta^-(bprog\ v_1, \cdots, v_i)$$

$$\dfrac{\dfrac{}{\vdash \mathcal{K}_1 : \cdot \Downarrow l(pat\ v_1, \cdots, v_i)^\perp}\,[\mathrm{I}] \quad \dfrac{\vdash \mathcal{K}_2 : \cdot \Uparrow \delta^-(bprog\ v_1, \cdots, v_i)}{\vdash \mathcal{K}_2 : \cdot \Downarrow \delta^-(bprog\ v_1, \cdots, v_i)}\,[\mathrm{R}\Downarrow]}{\dfrac{\vdash \mathcal{K} : \cdot \Downarrow l(pat\ v_1, \cdots, v_i)^\perp\ \otimes\ \delta^-(bprog\ v_1, \cdots, v_i)}{\vdash \mathcal{K} : \cdot \Downarrow \mathbf{unload}_i\ l\ pat\ bprog}\,[\mathrm{def}\Downarrow]}\,[\otimes]$$

O comando $\mathbf{unload}_i$ encontra uma tupla $(v_1, ..., v_i)$ com i elementos no location l tal que ela satisfaça o padrão *pat*. A execução é continuada com um *bprog*, que pode ser um *prog* ou um $\lambda var.bprog$. No segundo caso, a tupla $(v_1, ..., v_i)$ encontrada é passada como parâmetro para instanciar *var*. A escolha das variáveis envolve uma busca no location l, portanto é possível que sejam encontradas falhas nessa busca e seja necessário realizar *backtracking* durante essa prova.

Observe que ao aplicar a regra $\otimes$ é necessário dividir o contexto $\mathcal{K}$ em $\mathcal{K}_1$ e $\mathcal{K}_2$. Para que a regra I possa ser aplicada no *branch* à esquerda, é necessário que $\mathcal{K}_1[l] = \{l(pat\ v_1, ..., v_i)\}$ e $\mathcal{K}_1[k] = \emptyset,\ \forall\ k \neq l$. Isso significa que $\mathcal{K}_2$ conterá todas as outras fórmulas, e o programa continuará a execução com uma nova função $\mathcal{K}$ que não contém a tupla $(v_1, ..., v_i)$ no subexponencial l. No momento da aplicação da regra inicial, é feita a unificação das variáveis $v_1, v_2, ..., v_i$ com elementos que estão em l.

5.3.2.3 Comando `while`

$$\begin{gathered}\textbf{while}\ (t_1 \blacktriangledown t_2)\ (\lambda K.prog_1)\ prog_2 \triangleq \\ [(t_1 \blacktriangledown t_2)\ \otimes\ \delta^-((\lambda K.prog_1)(\textbf{while}\ (t_1 \blacktriangledown t_2)\ (\lambda K.prog_1)\ prog_2))] \\ \oplus\ [(t_1 \tilde{\blacktriangledown} t_2)\ \otimes\ \delta^-(prog_2)]\end{gathered}$$

$$\dfrac{\dfrac{\vdots}{\vdash \mathcal{K} : \cdot \Downarrow [(t_1 \blacktriangledown t_2)\ \otimes\ \delta^-((\lambda K.prog_1)(\textbf{while}\ (t_1 \blacktriangledown t_2)\ (\lambda K.prog_1)\ prog_2))] \oplus\ [(t_1 \tilde{\blacktriangledown} t_2)\ \otimes\ \delta^-(prog_2)]}\ [\oplus_i]}{\vdash \mathcal{K} : \cdot \Downarrow \textbf{while}\ (t_1 \blacktriangledown t_2)\ (\lambda K.prog_1)\ prog_2}\ [\text{def}\Downarrow]$$

Para a continuação dessa prova, é preciso aplicar a regra $\oplus_i$ e escolher um dos dois termos desse operador para continuar. A escolha que deve ser feita, semanticamente, é se o **while** irá iterar mais uma vez ($\oplus_1$) ou se a condição de parada já foi atingida e fluxo sai do **while** ($\oplus_2$). Abaixo são apresentadas as duas opções:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\vdash \cdot : \cdot \Downarrow t_1 \blacktriangledown t_2} \quad \dfrac{\vdash \mathcal{K} : \cdot \Uparrow \delta^-((\lambda K.prog_1)(\textbf{while}\ (t_1 \blacktriangledown t_2)\ (\lambda K.prog_1)\ prog_2}{\vdash \mathcal{K} : \cdot \Downarrow \delta^-((\lambda K.prog_1)(\textbf{while}\ (t_1 \blacktriangledown t_2)\ (\lambda K.prog_1)\ prog_2}\ [\text{R}\Downarrow]}{\vdash \mathcal{K} : \cdot \Downarrow (t_1 \blacktriangledown t_2)\ \otimes\ \delta^-((\lambda K.prog_1)(\textbf{while}\ (t_1 \blacktriangledown t_2)\ (\lambda K.prog_1)\ prog_2))}\ [\otimes]}{\vdots}\ [\oplus_1]$$

$$\dfrac{\dfrac{\dfrac{\vdots}{\vdash \cdot : \cdot \Downarrow (t_1 \tilde{\blacktriangledown} t_2)} \quad \dfrac{\vdash \mathcal{K} : \cdot \Uparrow \delta^-(prog_2)}{\vdash \mathcal{K} : \cdot \Downarrow \delta^-(prog_2)}\ [\text{R}\Downarrow]}{\vdash \mathcal{K} : \cdot \Downarrow (t_1 \tilde{\blacktriangledown} t_2)\ \otimes\ \delta^-(prog_2)}\ [\otimes]}{\vdots}\ [\oplus_2]$$

Em ambas as árvores de prova acima, o *branch* do lado esquerdo irá desenvolver a expressão que representa a condição para continuação do **while**, com a diferença que na segunda árvore é desenvolvido a negação dessa expressão[3]. O sucesso nessa computação significa que o *branch* direito pode ser executado normalmente. Caso ela falhe, haverá um *backtracking* até a aplicação da regra $\oplus_i$, o que significa que a execução do *branch* direito também será abortada. Observe que a computação pode falhar várias vezes se não houver uma heurística para escolher qual regra será aplicada e $\oplus_2$ for escolhida enquanto $(t_1 \blacktriangledown t_2)$ for válido. O símbolo $\blacktriangledown$ representa um dos operadores de comparação $\{\leq, <, =, >, \geq\}$, que serão aplicados aos termos t_1 e t_2 (que deverão ser numéricos). Uma discussão sobre a definição dessas operações pode ser encontrada na Seção 5.3.3.1.

5.3.2.4 Comando `loop`

$$\begin{gathered}\textbf{loop}_i\ l\ kprog\ prog \triangleq \\ [l(v_1, \cdots, v_i)^\perp\ \otimes\ \delta^-((kprog\ v_1, \cdots, v_i)\ \textbf{loop}_i\ l\ kprog\ prog)] \oplus\ !^{\hat{l}}\ (prog)\end{gathered}$$

[3] O operador $\tilde{\blacktriangledown}$ é o complementar do operador $\blacktriangledown$.

$$\dfrac{\dfrac{\vdots}{\vdash \mathcal{K} : \cdot \Downarrow [l(v_1, \cdots, v_i)^{\perp} \otimes \delta^{-}((kprog\ v_1, \cdots, v_i)\ \mathbf{loop}_i\ l\ kprog\ prog)] \oplus\ !^{\hat{l}}\ (prog)}\ [\oplus_i]}{\vdash \mathcal{K} : \cdot \Downarrow \mathbf{loop}_i\ l\ kprog\ prog}\ [\text{def}\Downarrow]$$

Para continuar a execução do comando $\mathbf{loop}_i$ é necessário escolher entre dois caminhos: (1) realizar mais uma iteração ou (2) terminar as repetições do *loop* e seguir o resto do programa. A primeira opção é equivalente à aplicação da regra $\oplus_1$, e a segunda, à aplicação da regra $\oplus_2$. Abaixo estão as árvores de prova para cada uma das possibilidades:

$$\dfrac{\dfrac{\dfrac{}{\vdash \mathcal{K}_1 : \cdot \Downarrow l(v_1, \cdots, v_i)^{\perp}}\ [\text{I}] \quad \dfrac{\vdash \mathcal{K}_2 : \cdot \Uparrow \delta^{-}((kprog\ v_1, \cdots, v_i)\ \mathbf{loop}_i\ l\ kprog\ prog)}{\vdash \mathcal{K}_2 : \cdot \Downarrow \delta^{-}((kprog\ v_1, \cdots, v_i)\ \mathbf{loop}_i\ l\ kprog\ prog)}\ [\text{R}\Downarrow]}{\vdash \mathcal{K} : \cdot \Downarrow l(v_1, \cdots, v_i)^{\perp} \otimes \delta^{-}((kprog\ v_1, \cdots, v_i)\ \mathbf{loop}_i\ l\ kprog\ prog)}\ [\otimes]}{\vdots}\ [\oplus_1]$$

$$\dfrac{\dfrac{\vdash \mathcal{K}_{\leq \hat{l}} : \cdot \Uparrow prog}{\vdash \mathcal{K} : \cdot \Downarrow !^{\hat{l}}\ (prog)}\ [!^{\hat{l}}]}{\vdots}\ [\oplus_2]$$

Na expressão que define o comando $\mathbf{loop}_i$, *kprog* é um programa que recebe como parâmetro i variáveis e uma continuação. Ele é utilizado somente quando o programa ainda deve iterar mais uma vez no *loop*, ou seja, é o bloco que será repetido. As variáveis que são passadas são aquelas escolhidas do location l, e a continuação é o próprio comando **loop**, para que ele seja executado novamente.

Esse comando irá iterar nos elementos do location l, retirando-os, um a um, e processando-os. O programa só irá continuar (sair do *loop*) se l estiver vazio[4], ou seja, se todos os elementos tenham sido processados. Caso a regra $\oplus_2$ tenha sido escolhida antes que esse location esteja vazio, irá ocorrer uma falha na aplicação da regra $!^{\hat{l}}$ e será realizado o *backtracking* para que $\oplus_1$ seja escolhido.

5.3.2.5 Comando `new`

$$\mathbf{new}\ loc\ \lambda L.prog\ \triangleq\ \Cap_l\ loc\ prog$$

$$\dfrac{\dfrac{\mathcal{L} \cup \{loc\} \vdash \mathcal{K} : \cdot \Uparrow prog}{\mathcal{L} \vdash \mathcal{K} : \cdot \Uparrow \Cap_l loc\ prog}\ [\Cap_l]}{\mathcal{L} \vdash \mathcal{K} : \cdot \Uparrow \mathbf{new}\ loc\ \lambda L.prog}\ [\text{def}\Uparrow]$$

O comando **new** é utilizado para criar novos locations durante a execução do programa. Ele pode ser visto como uma maneira de alocar espaços para conjunto de dados sob demanda. Nessa regra, $\mathcal{L}$ é o conjunto de locations e *loc* é um novo location instanciado no programa.

[4]É importante ressaltar que nesse *loop* os elementos são de fato retirados de um *location* e não são colocados de volta.

Esse novo *location* será linear e estará relacionado somente com o inf (*location* para as fórmulas clássicas), do qual ele é menor ($loc \preceq \text{inf}$).

5.3.2.6 Comando `if then else`

$$\textbf{if}\ (t_1 \blacktriangledown t_2)\ \textbf{then}\ kprog_1\ \textbf{else}\ kprog_2\ prog \triangleq$$
$$((t_1 \blacktriangledown t_2) \otimes \delta^-(kprog_1 prog)) \oplus ((t_1 \tilde{\blacktriangledown} t_2) \otimes \delta^-(kprog_2 prog))$$

$$\dfrac{\dfrac{\vdots}{\vdash \mathcal{K} : \cdot \Downarrow ((t_1 \blacktriangledown t_2) \otimes \delta^-(kprog_1 prog)) \oplus ((t_1 \tilde{\blacktriangledown} t_2) \otimes \delta^-(kprog_2 prog))}\,[\oplus_i]}{\vdash \mathcal{K} : \cdot \Downarrow \textbf{if}\ (t_1 \blacktriangledown t_2)\ \textbf{then}\ kprog_1\ \textbf{else}\ kprog_2\ prog}\,[\text{def}\Downarrow]$$

Nesse momento deve ser feita uma escolha não-determinística para seguir a prova. Abaixo estão as duas possibilidades:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\cdot : \cdot \Downarrow t_1 \blacktriangledown t_2} \quad \dfrac{\mathcal{K} : \cdot \Uparrow \delta^-(kprog_1 prog)}{\mathcal{K} : \cdot \Downarrow \delta^-(kprog_1 prog)}\,[\text{R}\Downarrow]}{\mathcal{K} : \cdot \Downarrow (t_1 \blacktriangledown t_2) \otimes \delta^-(kprog_1 prog)}\,[\otimes]}{\vdots}\,[\oplus_1]$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\cdot : \cdot \Downarrow t_1 \tilde{\blacktriangledown} t_2} \quad \dfrac{\mathcal{K} : \cdot \Uparrow \delta^-(kprog_2 prog)}{\mathcal{K} : \cdot \Downarrow \delta^-(kprog_2 prog)}\,[\text{R}\Downarrow]}{\mathcal{K} : \cdot \Downarrow (t_1 \tilde{\blacktriangledown} t_2) \otimes \delta^-(kprog_2 prog)}\,[\otimes]}{\vdots}\,[\oplus_2]$$

Este comando **if** utiliza uma comparação como condição para seguir o programa. Caso a condição não seja verdadeira e a regra $[\oplus_1]$ tenha sido escolhida, ocorrerá um *backtracking* até a aplicação da regra $[\oplus]$ e a outra opção será escolhida. Como as condições para continuação da prova em ambos os casos são mutuamente exclusivas, garante-se que a execução certamente irá continuar com o bloco que deve ser executado ($kprog_1$ ou $kprog_2$) recebendo como continuação o resto do programa (*prog*).

Observe que a aplicação da regra $\otimes$ exige a divisão do contexto $\mathcal{K}$. Nessa regra podemos seguir com todo o contexto na prova à direita, já que a prova de $(t_1 \blacktriangledown t_2)$ não consome fórmula alguma.

5.3.2.7 Comando `if then else` com `is_empty`

$$\textbf{if}\ is\_empty\ l\ \textbf{then}\ kprog_1\ \textbf{else}\ kprog_2\ prog \triangleq (!^{\hat{l}}(kprog_1 prog)) \oplus (\exists x.l(x) \parr \delta^-(kprog_2 prog))$$

$$\dfrac{\dfrac{\vdots}{\vdash \mathcal{K} : \cdot \Downarrow (!^{\hat{l}}(kprog_1 prog)) \oplus (\exists x.l(x) \parr \delta^-(kprog_2 prog))}\,[\oplus_i]}{\vdash \mathcal{K} : \cdot \Downarrow \textbf{if}\ is\_empty\ l\ \textbf{then}\ kprog_1\ \textbf{else}\ kprog_2\ prog}\,[\text{def}\Downarrow]$$

Nesse momento deverá ser feita uma escolha não-determinística para seguir a prova. Abaixo apresentamos as duas possibilidades:

$$\dfrac{\dfrac{\dfrac{\vdots}{\mathcal{K}_{\leq \hat{l}} : \cdot \Uparrow kprog_1 prog}}{\mathcal{K} : \cdot \Downarrow !^{\hat{l}}(kprog_1 prog)}\,[!^{\hat{l}}]}{\vdots}\,[\oplus_1]$$

$$\dfrac{\dfrac{\dfrac{\mathcal{K} : \cdot \Downarrow l(x)\{t/x\}, \delta^-(kprog_2 prog)}{\mathcal{K} : \cdot \Downarrow \exists x.l(x), \delta^-(kprog_2 prog)}\,[\exists]}{\mathcal{K} : \cdot \Downarrow \exists x.l(x) \otimes \delta^-(kprog_2 prog)}\,[\otimes]}{\vdots}\,[\oplus_2]$$

Se a escolha for seguir pela aplicação da regra $[\oplus_1]$, a prova só terá sucesso se l de fato estiver vazio. Caso contrário, a aplicação da regra $[!^{\hat{l}}]$ irá falhar e será realizado um *backtracking* até o ponto onde houve a escolha não-determinística, e o outro caminho será tomado.

Se a escolha for pela aplicação da regra $[\oplus_2]$, a prova só terá sucesso caso exista de fato algum elemento no *location* l, o que fará com $\exists x.l(x)$ seja verdadeiro. Caso contrário, será realizado um *backtracking* até a aplicação da regra $[\oplus_2]$ e a outra opção será escolhida.

Em ambas as situações, o programa irá executar o bloco de código correto ($kprog_1$ ou $kprog_2$, dependendo se o contexto l está vazio ou não) que recebe como continuação o restante do programa ($prog$).

5.3.2.8 Comando para escolha de continuação

$$prog_1 \,[]\, prog_2 \triangleq prog_1 \oplus prog_2$$

$$\dfrac{\dfrac{\vdots}{\vdash \mathcal{K} : \cdot \Downarrow prog_1 \oplus prog_2}\,[\oplus_i]}{\vdash \mathcal{K} : \cdot \Downarrow prog_1 \,[]\, prog_2}\,[\text{def}\Downarrow]$$

A utilização desse comando faz com que a computação execute um dos dois programas. Se um deles falhar, é feito um *backtracking* até a aplicação da regra $\oplus_i$ e escolhe-se a outra opção.

Essa é uma maneira alternativa de fazer um **if** quando a condição não for comparação aritmética ou conteúdo de um location. Por exemplo, suponha um algoritmo em grafo no qual, em certo momento, deve-se escolher que ação tomar baseando-se no fato do vértice já ter sido visitado ou não. Podemos escrever o seguinte código:

$$\begin{array}{l}\mathbf{unload}_2\ l\ \langle v, visitado\rangle \\ \quad \text{ação para vértice visitado} \\ []\mathbf{unload}_2\ l\ \langle v, naoVisitado\rangle \\ \quad \text{ação para vértice não visitado}\end{array}$$

Nessa situação, l deve ser um location que armazena os vértices com a informação de visita. Considerando a semântica do comando [], o fluxo do programa irá escolher continuar por um dos dois comandos **unload**. Caso a escolha seja pelo primeiro e não exista vértice v visitado, a computação falha e realiza um *backtracking* que irá escolher o segundo **unload**. Observe que a variável v não está instanciada. Esse trecho de código irá escolher um vértice v (instanciar) no momento em que realizar o **unload**.

5.3.2.9 Comando `end`

$$\mathbf{end} \triangleq \top$$

$$\dfrac{\dfrac{}{\vdash \mathcal{K} : \cdot \Uparrow \top}\,[\top]}{\vdash \mathcal{K} : \cdot \Uparrow \mathbf{end}}\,[\text{def}\Uparrow]$$

O comando **end** determina o final de uma computação. A utilização de $\top$ permite que se termine a prova.

5.3.3 Aritmética

A aritmética da linguagem consiste apenas de operações com números inteiros e positivos. Como elementos básicos temos o zero, representado por *zero*, e a função sucessor, representada por *succ*. Dessa maneira, o número 1 (um) será escrito como *succ zero* durante uma prova.

5.3.3.1 Operadores de comparação

A linguagem BAG possui cinco operadores de comparação aritmética: $\leq, <, =, >, \geq$. Com exceção da igualdade, eles podem ser definidos através de fórmulas da lógica linear, como apresentado abaixo:

$$\begin{aligned}
x \leq y &\triangleq [x = zero] \oplus [\exists x'y'.\ (x = succ\ x') \otimes (y = succ\ y') \otimes (x' \leq y')] \\
x < y &\triangleq [x \leq y] \otimes [x = y]^{\perp} \\
x > y &\triangleq [x \geq y] \otimes [x = y]^{\perp} \\
x \geq y &\triangleq [y = zero] \oplus [\exists x'y'.\ (x = succ\ x') \otimes (y = succ\ y') \otimes (x' \geq y')]
\end{aligned}$$

Essas definições são todas recursivas e seguem o mesmo raciocínio. Suponha que queremos saber se $x < y$. Então, encontram-se x' e y' antecessores de x e y e aplica-se a estes o mesmo operador. Se for obtida uma expressão em que x seja igual a *zero*, garantimos que ele é o menor. Se y atingir *zero* antes de x, não existirá antecessor e ambas as expressões que constituem o $\oplus$ (disjunção) irão falhar (serão falsas), e portanto a expressão $x < y$ será falsa.

As definições das comparações são fórmulas puramente positivas, pois utilizam somente operadores síncronos: $\oplus$, $\otimes$ e $\exists$. Dessa maneira, a prova de $x \blacktriangledown y$ ($\blacktriangledown \in \{\leq, <, >, \geq\}$) é composta por exatamente uma fase positiva (síncrona) e não consome fórmulas do contexto (o que só ocorre em uma fase assíncrona ou na mudança de uma fase assíncrona para síncrona - ver regras D e R$\Uparrow$).

Entretanto, como essas operações já existem em λ-Prolog, elas não foram implementadas novamente. A prova de cada uma dessas comparações foi feita com o axioma inicial de *SELLF*, apenas exigindo que a mesma comparação fosse verdadeira em λ-Prolog.

5.3.4 Exemplo

O programa em BAG da Figura 5.13 ordena um vetor de números inteiros e positivos. Como o contexto da lógica linear é um conjunto, e portanto não distingue entre permutações diferentes dos elementos, foi utilizado um par $[i, n]$ para representar cada elemento do vetor, onde i é o índice e n o valor. Os comandos *loop2* e *unload2* são utilizados para lidar com os pares do contexto.

O algoritmo apresentado é similar à ordenação por seleção, que escolhe o menor elemento do vetor e o coloca na primeira posição do sub-vetor ainda não ordenado. Inicialmente, o vetor não ordenado está em $l1$. Se esse vetor é $[4, 3, 2, 1]$, ele é representado da seguinte forma: $[1, 4], [2, 3], [3, 2], [4, 1]$. O primeiro loop escolhe o primeiro elemento de $l1$, ou seja, $[1, 4]$ no exemplo dado, e o armazena em $l2$. É importante observar que um comando *loop* retira os elementos do *location*, portanto, o segundo loop vai percorrer todos os outros elementos de $l1$ verificando se existe algum menor do que aquele que acabou de ser colocado em $l2$. Caso exista, apenas os elementos (não os índices) são trocados. Após o segundo loop então, $[1, 1]$ estará em $l2$ e $[2, 3], [3, 2], [4, 4]$ estará em $l3$. Os elementos já verificados são colocados em $l3$, e logo após o término dessa etapa são colocados de volta em $l1$ para que o programa continue buscando o próximo elemento menor.

```
loop2 l1 ( ( (
    load [N,D] l2 (
        loop2 l1 ( (
            unload2 l2 ( (
                if (Dx < Dmin) (
                    load [Min,Dx] l2 (load [X,Dmin] l3 K1)
                )
                (
                    load [Min,Dmin] l2 (load [X,Dx] l3 K1)
                )
                K
            ))
        ))
        (
            loop2 l3 ( ( load [A,B] l1 K3)) K1
        )
    )
)))
done
```

Figura 5.13. Ordenação de um vetor em BAG.

Capítulo 6

Conclusão

6.1 Resultados

A primeira contribuição desse trabalho foi a implementação de um interpretador para a lógica linear com subexponenciais, que não existia até o momento. Com essa implementação foi possível validar as especificações de outros sistemas nessa lógica.

A segunda contribuição foi a especificação da linguagem de programação BAG. Ela foi proposta por Nigam [Nigam, 2009], porém, durante a implementação foram encontradas e corrigidas algumas falhas, por exemplo, a semântica do comando **if**. Além disso, foram apresentadas as derivações das definições de todos os comandos, provando assim que a execução de um programa em BAG corresponde sempre a uma prova na lógica linear com subexponenciais. Mais do que isso, com a utilização de *delays* é possível dividir a árvore de prova do programa em fases que delimitam a execução de cada comando separadamente. Esse resultado permite que se explore novas utilizações da lógica linear com subexponenciais. A possibilidade de especificar qualquer algoritmo sequencial na lógica torna possível que algumas propriedades desses algoritmos sejam verificadas a partir de sua descrição formal. O interpretador de *SELLF* pode ser utilizado para realizar essas provas automaticamente.

6.2 Trabalhos futuros

Apesar de apresentar uma implementação para *SELLF* nesse trabalho, muitas melhorias ainda podem ser feitas. Uma delas é a implementação desse sistema como um interpretador interativo, como existe para a linguagem Prolog e λ-Prolog, por exemplo. Esse sistema está atualmente em fase de implementação, na linguagem OCaml, em parceria com Vivek Nigam e Elaine Pimentel. Espera-se obter um *framework* similar ao da linguagem λ-Prolog, estendido com os conectivos da lógica linear com subexponenciais. Dessa maneira sua utilização ficará independente da construção manual do sequente e chamada a um predicado (`sellf`). As especificações serão descritas de maneira direta e toda a tradução e aplicação de definições será transparente para o usuário.

Outro ponto a ser explorado são quais informações podem ser extraídas de uma implementação em *SELLF*. Foram estudadas as especificações de sistemas lógicos na lógica linear comum e sabe-se que é possível provar a propriedade de *cut-elimination*[1] automaticamente com

[1] A regra *cut* é admissível.

essas especificações [Miller & Pimentel, 2002]. Em particular, foi apresentada uma maneira automática para se provar essa propriedade para a lógica LU[2] [Pimentel & Miller, 2005]. Essa prova foi implementada no sistema *SELLF* apresentado neste trabalho. Seria interessante estender esse resultado para especificações na lógica linear com subexponenciais, já que ela consegue codificar um conjunto maior de sistemas.

[2] *Logic of Unicity* [Girard, 1991]

Apêndice A

Código-fonte

Nesse apêndice estão os códigos-fonte de todas as implementações citadas neste trabalho. Cada um foi implementado como um módulo de λ-Prolog, que é formado por dois arquivos. O primeiro é a assinatura, onde estão a declaração dos tipos utilizados, construtores de tipos, constantes e operadores. O segundo é a implementação dos predicados declarados na assinatura.

A.1 Implementação de *SELLF*

A.1.1 Assinatura

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                   %
% Interpreter for the linear logic with subexponentials (SELLF)     %
% Signature file.                                                   %
%                                                                   %
% Author: Giselle Machado N. Reis                                   %
%                                                           2010    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

sig sellf.

%% Top level predicate
type sellf (list k) -> (list o) -> (list o) -> o.

%% Type definitions

kind pair type -> type -> type. %% pair type constructor
kind k type.
type k_elm index -> list E -> k.

kind index type.  % Subexponential index type

type at index -> A -> o.

%%%% The logical connectives for SELLF
%% Some are commented out because they already exist in Teyjus

%% Asynchronous operatos

%type    &       o -> o -> o.
 type    @       o -> o -> o.
 type    ?       index -> A -> o. % with index
```

```
%type    pi      o -> o.
 type    top     o.   % a kind of true
 type    bottom  o.   % a kind of false
 type    create  index -> o -> o.

 infixl  @     8.

%% Synchronous operators

%type    ;       o -> o -> o.
%type    ,       o -> o -> o.
%type    sigma  o -> o.
 type    bang    index -> o -> o. % with index
 type bang_empty index -> o -> o. % with index
%type    true     o.   % a kind of true
 type    zero    o.   % a kind of false (never used in the inference rules... why?)
 type    inst    index -> o -> o.

%% Function type definitions

type prove_a (list k) -> (list o) -> (list o) -> int -> o.
type prove_s (list k) -> (list o) -> o -> int -> o.

% This is necessary so that I can have other types in K, like integers
type my_not A -> A.

type pos            o -> o.
type neg            o -> o.
type non_literal    o -> o.
type literal        o -> o.
type cont           index -> o.
type weak           index -> o.
type pred           index -> index -> o.

%% Clauses for functions involving K
type k_geq              index -> (list k) -> (list k) -> o.
type k_subset           list index -> (list k) -> (list k) -> o.
type k_subset2          list index -> (list k) -> (list k) -> o.
type k_insert           index -> A -> (list k) -> (list k) -> o.
type k_split            (list k) -> (list k) -> (list k) -> o.
type k_get              index -> (list k) -> list A -> o.
type k_not_weakenable   (list k) -> list A -> o.
type k_insert_end       index -> A -> (list k) -> (list k) -> o.

%% Clauses of side conditions
type condition1         (list k) -> o.
type condition_dl       index -> o.
type not_condition_dl   index -> o.
type condition_and      (list k) -> (list k) -> o.
type condition_bang     index -> (list k) -> o.
type condition_init     (list k) -> list o -> A -> o.
type condition_init_2   (list k) -> list o -> index -> A -> o.

%% Auxiliars
type is_in          A -> list A -> o.
type is_in_k        A -> (list k) -> o.
type is_index       index -> (list k) -> o.
type has_index      A -> index -> (list k) -> o.
type is_empty       index -> (list k) -> o.
type append         list A -> list A -> list A -> o.
type split list     A -> list A -> list A -> o.
type insert_nd      A -> list A -> list A -> o.

%% I/O
kind printcommand    type.
```

```
type pc              A -> printcommand.
type nl, fl          printcommand.
type printf          list printcommand -> o.
type print_context   (list k) -> (list o) -> o.
type print_k         (list k) -> o.
type print_g         (list o) -> o.
```

A.1.2 Código-fonte

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                              %
% Interpreter for the linear logic with subexponentials (SELLF)                %
%                                                                              %
% Author: Giselle Machado N. Reis                                              %
%                                                                        2010  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% OBS: Since there are a lot of calls to functions that are not acctually part of the rule,
%% it was conventioned that these calls be placed one tab ahead from the actual implementation
%% of the rule.

%% OBS2: the integer N in each rule is for debugging purposes.

module sellf.

%%%% Inference rules for SELLF

sellf K G L :- prove_a K G L 0.

%%% Asynchronous Phase

prove_a K G (top::L) N :-
        printf [nl, pc N, pc " Top rule.", nl, fl],
        print_context K G.

prove_a K G (bottom::L) N :-
        printf [nl, pc N, pc " Bottom rule: continuing with list '", pc L, nl, fl],
        N1 is N + 1,
    prove_a K G L N1.

prove_a K G ((A & B)::L) N :-
        printf [nl, pc N, pc " Add and rule: divided in two branches with '", pc A,
            pc "' and '", pc B, pc "'", nl, fl],
        N1 is (N * 100 + 1),
        N2 is (N * 100 + 2),
    prove_a K G (A::L) N1, prove_a K G (B::L) N2.

prove_a K G ((A @ B)::L) N :-
        printf [nl, pc N, pc " Mult or rule: '", pc A, pc "' and '", pc B,
            pc "' put in the beginning of the list.", nl, fl],
        N1 is N + 1,
    prove_a K G (A::B::L) N1.

prove_a K G ((? I A)::L) N :-
    k_insert I A K K_new,
        printf [nl, pc N, pc " Question mark rule: inserting '", pc A,
            pc "' in subexp '", pc I, pc "'", nl, fl],
        N1 is N + 1,
    prove_a K_new G L N1.

prove_a K G ((pi A)::L) N :-
        printf [nl, pc N, pc " For all rule: with expression '", pc A, pc "'", nl, fl],
        N1 is N + 1,
    pi X\ prove_a K G ((A X)::L) N1.
```

```
%% Location creation
prove_a K G ((create I A)::L) N :-
        printf [nl, pc N, pc  " Creating new subexponential index: ", pc I, nl, fl],
        N1 is N + 1,
    prove_a ((k_elm I nil)::K) G (A::L) N1.

%%% Synchronous Phase

prove_s K nil (true) N :-
        printf [nl, pc N, pc " True rule.", nl, fl],
    condition1 K. % side condition: K[I-W] is empty

prove_s K G (A ; B) N :-
        printf [nl, pc N, pc " Add or 1 rule: chose to prove'", pc A, pc "'", nl, fl],
        N1 is N + 1,
    prove_s K G A N1.

prove_s K G (A ; B) N :-
        printf [nl, pc N, pc " Add or 2 rule: chose to prove '", pc B, pc "'", nl, fl],
        N1 is N + 1,
    prove_s K G B N1.

prove_s K G (A , B) N :-
        printf [nl, pc N, pc " Mult and rule: proving '", pc A, pc "' and '", pc B, pc "'", nl, fl],
    k_split K K1 K2,
    split G G1 G2,
    condition_and K1 K2,
        N1 is (N * 100 + 1),
        N2 is (N * 100 + 2),
    prove_s K1 G1 A N1,
    prove_s K2 G2 B N2.

prove_s K G (sigma A) N :-
        N1 is N + 1,
    sigma X\ prove_s K G (A X) N1.

prove_s K nil (bang I A) N :-
    condition_bang I K,
    k_geq I K K1,
        printf [nl, pc N, pc " Bang rule: with term '", pc A, pc "' and subexp '", pc I, pc "'", nl, fl],
        N1 is N + 1,
    prove_a K1 nil (A::nil) N1.

% making things easier =)
prove_s K nil (bang_empty I A) N :-
    is_empty I K,
        printf [nl, pc N, pc " Bang empty rule: with term '", pc A, pc "' and subexp '", pc I, pc "'", nl, fl],
        N1 is N + 1,
    prove_a K nil (A::nil) N1.

%% Location instantiation
prove_s K G (inst I A) N :-
        printf [nl, pc N, pc " Instantiating subexponential index: ", pc I, nl, fl],
        N1 is N + 1,
    sigma X\ (is_index X K, prove_s K G ((I\ A) X) N1).

%%% Reaction, Identity and Decide rules

%% R - arrow down
prove_s K G N Num:-
    neg N,
        printf [nl, pc Num, pc " R arrow down rule: found '", pc N,
            pc "' negative, changing to asynchronous phase.", nl, fl],
        Num1 is Num + 1,
```

```
    prove_a K G (N::nil) Num1.

%% R - arrow up
prove_a K G (P::L) N :-
    (pos P ; literal P),
        printf [nl, pc N, pc" R arrow up rule: passing positive or literal '", pc P,
            pc "' to linear (Gamma) context. Continuing with '", pc L, pc "'", nl, fl],
        N1 is N + 1,
    prove_a K (P::G) L N1.

%% D1
prove_a K G nil N :-
    insert_nd P G1 G,
    (pos P ; non_literal P),
        printf [nl, pc N, pc " D1 rule: choosing '", pc P,
            pc "' positive from linear (Gamma) context, changing to synchronous phase.", nl, fl],
        N1 is N + 1,
    prove_s K G1 P N1.

%% Di - I can suffer contraction and weakening
prove_a K G nil N :-
    condition_dl I,
    k_insert I P K1 K,
    % Using this to avoid loops: everytime a formula is chosen, it is put in the end of the list.
    k_insert_end I P K1 K2,
    (pos P ; non_literal P),
        printf [nl, pc N, pc " Di 1 rule: choosing '", pc P,
            pc "' positive (or formula) from subexp '", pc I,
            pc "', changing to synchronous phase.", nl, fl],
        N1 is N + 1,
    prove_s K2 G P N1.

%% Di - I can't suffer contraction or weakening
prove_a K G nil N :-
    not_condition_dl I,
    k_insert I P K1 K,
    (pos P ; non_literal P),
        printf [nl, pc N, pc " Di 2 rule: choosing '", pc P,
            pc "' positive (or formula) from subexp '", pc I,
            pc "', changing to synchronous phase.", nl, fl],
        N1 is N + 1,
    prove_s K1 G P N1.

%% I
prove_s K G A N :-
    condition_init K G A,
        printf [nl, pc N, pc " Initial rule: proving '", pc A, pc "'", nl, fl],
        print_context K G,
        flush std_out.

% Initial rule, version for when the atom is of the form l(m)
prove_s K G (at I A) N :-
    % (at I A) is positive and literal by definition
    condition_init_2 K G I A,
        printf [nl, pc N, pc " Initial rule v2: proving '", pc I, pc "(", pc A, pc ")", nl, fl],
        print_context K G,
        flush std_out.

% Initial rule, versions for arithmetical comparisson
prove_s K nil (A <= B) N :-
    k_not_weakenable K nil, A <= B,
        printf [nl, pc N, pc " Initial rule arith. comp. proving '", pc A, pc " <= ", pc B, pc "'", nl, fl],
        print_context K G,
        flush std_out.
prove_s K nil (A < B) N :-
```

```
    k_not_weakenable K nil, A < B,
        printf [nl, pc N, pc " Initial rule arith. comp. proving '", pc A, pc " < ", pc B, pc "'", nl, fl],
        print_context K G,
        flush std_out.
prove_s K nil (A > B) N :-
    k_not_weakenable K nil, A > B,
        printf [nl, pc N, pc " Initial rule arith. comp. proving '", pc A, pc " > ", pc B, pc "'", nl, fl],
        print_context K G,
        flush std_out.
prove_s K nil (A >= B) N :-
    k_not_weakenable K nil, A >= B,
        printf [nl, pc N, pc " Initial rule arith. comp. proving '", pc A, pc " >= ", pc B, pc "'", nl, fl],
        print_context K G,
        flush std_out.
% Initial rule, versions for arithmetical comparisson negated
prove_s K nil (not (A <= B)) N :-
    k_not_weakenable K nil, not (A <= B),
        printf [nl, pc N, pc " Initial rule arith. comp. proving 'not ", pc A, pc " <= ", pc B, pc "'", nl, fl],
        print_context K G,
        flush std_out.
prove_s K nil (not (A < B)) N :-
    k_not_weakenable K nil, not (A < B),
        printf [nl, pc N, pc " Initial rule arith. comp. proving 'not ", pc A, pc " < ", pc B, pc "'", nl, fl],
        print_context K G,
        flush std_out.
prove_s K nil (not (A > B)) N :-
    k_not_weakenable K nil, not (A > B),
        printf [nl, pc N, pc " Initial rule arith. comp. proving 'not ", pc A, pc " > ", pc B, pc "'", nl, fl],
        print_context K G,
        flush std_out.
prove_s K nil (not (A >= B)) N :-
    k_not_weakenable K nil, not (A >= B),
        printf [nl, pc N, pc " Initial rule arith. comp. proving 'not ", pc A, pc " >= ", pc B, pc "'", nl, fl],
        print_context K G,
        flush std_out.

%%%% Operations of the function K

% Erases the formulas in indexes that are lower than or not related to i.
k_geq I nil nil.
k_geq I ((k_elm X L)::T) ((k_elm X L)::T2) :- pred I X, k_geq I T T2.
k_geq I ((k_elm X L)::T) ((k_elm X nil)::T2) :- not (pred I X), k_geq I T T2.

% Chooses the formulas with subexponential index in S.
k_subset S nil nil.
k_subset S ((k_elm X L)::T) ((k_elm X L)::T2) :- is_in X S, k_subset S T T2.
k_subset S ((k_elm X L)::T) K_new :- not (is_in X S), k_subset S T K_new.   % in case X is not in S !is_in X S

% Inserts an element in one of K's list. Associates a new element with an existing subexponential index.
k_insert I A nil nil :- fail. % ERROR: this index does not exist.
k_insert I A ((k_elm I L)::T) ((k_elm I L1)::T) :- insert_nd A L L1.
k_insert I A ((k_elm X L)::T) ((k_elm X L)::T2) :- k_insert I A T T2.

% Union of functions
k_split nil nil nil.
k_split ((k_elm I L)::T) ((k_elm I L1)::T1) ((k_elm I L2)::T2) :-
        not (cont I),
        split L L1 L2,
        k_split T T1 T2.
% OBS: duplicating elements that can suffer weakening *and* contraction. Assuming that cont is subset of weak
k_split ((k_elm I L)::T) ((k_elm I L)::T1) ((k_elm I L)::T2) :-
        cont I,
        weak I,
        k_split T T1 T2.
```

```
% Gets the formulas with subexponential I (not in thesis - implemented to be used in condition_and)
k_get I nil nil :- fail.
k_get I ((k_elm I L)::T) L.
k_get I ((k_elm J L)::T) S :- k_get I T S.

% Returns the set K[I-W] (not in thesis - implemented to be used in condition_init)
k_not_weakenable nil nil.
k_not_weakenable ((k_elm I nil)::T) L1 :- not (weak I), k_not_weakenable T L1.
k_not_weakenable ((k_elm I L)::T) L :- not (weak I), k_not_weakenable T nil.
k_not_weakenable ((k_elm I L)::T) L2 :- weak I, k_not_weakenable T L2.

% Inserts an element in the end of one of K's list. (not in thesis - trying to solve loops)
k_insert_end I A nil nil :- fail. % ERROR: this index does not exist.
k_insert_end I A ((k_elm I L)::T) ((k_elm I L1)::T) :- append L (A::nil) L1.
k_insert_end I A ((k_elm X L)::T) ((k_elm X L)::T2) :- k_insert_end I A T T2.


%%%% Side Conditions' expressions

% K[I\W] = empty (only the formulas that can suffer weakening are left - exactly because they be weakened)
condition1 nil.
condition1 ((k_elm I L)::T) :- weak I, condition1 T.
condition1 ((k_elm I L)::T) :- L = nil, condition1 T.

% I is in C and W
condition_dl I :- cont I, weak I.

% I is not in C and W at the same time
not_condition_dl I :- weak I, not (cont I).
not_condition_dl I :- cont I, not (weak I).
not_condition_dl I :- not (cont I), not (weak I).

% K1 = K2 in subexponentials that can suffer contraction and weakening
condition_and nil _.
condition_and ((k_elm I L)::T) K :-
        cont I,
        weak I,
        k_get I K L,
        condition_and T K.
condition_and ((k_elm I L)::T) K :-
% because I don't want this to be checked two times when both conditions are met.
        (not (cont I) ; not (weak I) ), !,
        condition_and T K.

% Every set with index not bigger than L (lower than or unrelated) which can't suffer weakening must be empty
condition_bang L nil.
condition_bang L ((k_elm I nil)::T) :- not (pred L I), not (weak I), condition_bang L T.
condition_bang L ((k_elm I F)::T) :- pred L I, condition_bang L T.
condition_bang L ((k_elm I F)::T) :- weak I, condition_bang L T.

% (not A) is in G union K[I] and G union K[I-W] is contained in {not A}  (for the initial axiom)
condition_init K G A :-
            pos A,
            literal A,
            ( G = ((my_not A)::nil) ; (is_in_k (my_not A) K, G = nil) ),
            (k_not_weakenable K nil ; k_not_weakenable K ((my_not A)::nil)).

condition_init_2 K G I A :-
            has_index (my_not A) I K , G = nil ,
            (k_not_weakenable K nil ; k_not_weakenable K ((my_not A)::nil)).

condition_init K G (my_not A) :-
            pos (my_not A),
            literal (my_not A),
            ( G = ((A)::nil) ; (is_in_k (A) K, G = nil) ),
```

```
            (k_not_weakenable K nil ; k_not_weakenable K ((A)::nil)).

condition_init_2 K G I (my_not A) :-
            has_index A I K , G = nil ,
            (k_not_weakenable K nil ; k_not_weakenable K ((A)::nil)).


%%%% Positive and negative formulas
pos (A , B).
pos (A ; B).
pos (bang I A).
pos (sigma A).
pos true.
pos zero.
pos (at I A). % l(m)
pos (my_not A) :- neg A.

pos (A <= B).
pos (A < B).
pos (A > B).
pos (A >= B).

neg (A & B).
neg (A @ B).
neg (? I A).
neg (pi A).
neg top.
neg bottom.
neg (my_not A) :- pos A.

%%%% Definition of literal
non_literal (A , B).
non_literal (A ; B).
non_literal (bang I A).
non_literal (sigma A).
non_literal (true).
non_literal (zero).
non_literal (A & B).
non_literal (A @ B).
non_literal (? I A).
non_literal (pi A).
non_literal (top).
non_literal (bottom).
literal A :- not (non_literal A).

%%%% Auxiliar functions

% is_in X L -> true if the element X is in the list L.
is_in X (X::T).
is_in X (Y::T) :- is_in X T.

% is_in_k A K -> version of the funcion above for K lists
is_in_k A ((k_elm I L)::T) :- is_in A L.
is_in_k A ((k_elm I L)::T) :- is_in_k A T.

% true if X is a subexponential index in K
is_index X ((k_elm X L)::T).
is_index X ((k_elm I L)::T) :- is_index X T.

% true if A has index I in K
has_index A I ((k_elm I L)::T) :- is_in A L.
has_index A I ((k_elm X L)::T) :- has_index A I T.

% true if index I is empty is k
is_empty I ((k_elm I nil)::T).
```

```
is_empty I ((k_elm X L)::T) :- is_empty I T.

append nil K K.
append (H1::T1) K (H1::T2) :- append T1 K T2.

split nil nil nil.
split (H::L) (H::L1) L2 :- split L L1 L2.
split (H::L) L1 (H::L2) :- split L L1 L2.

% non-deterministic insertion
insert_nd A L (A::L).
insert_nd A (H::L) (H::L1) :- insert_nd A L L1.


%%%% I/O

printf nil.
printf (pc (Item : string) :: L) :- !, print Item, printf L.
printf (pc Item :: L)            :- term_to_string Item Str, print Str, printf L.
printf (nl :: L)                 :- print "\n", printf L.
printf (fl :: L)                 :- flush std_out, printf L.

print_context K G :- print_k K, print_g G.
print_k K :- print "- K: ", term_to_string K StrK, print StrK, print "\n", flush std_out.
print_g G :- print "- G: ", term_to_string G StrG, print StrG, print "\n", flush std_out.
```

A.2 Implementação de *G1m*

A.2.1 Assinatura

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                              %
% Implementation of G1m's specification in LL using SELLF                      %
% Signature file.                                                              %
%                                                                              %
% Author: Giselle Machado N. Reis                                              %
%                                                                        2010  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

sig g1m.
accum_sig sellf.

kind form type.

%% Predicates
type g1m (list form) -> (list form) -> o.
type atomize_left (list form) -> list o -> o.
type atomize_right list form -> list o -> o.

%% Brackets down and up
type bd form -> o.
type bu form -> o.

%% Connectives
type imp    form -> form -> form.
type and    form -> form -> form.
type or     form -> form -> form.
type forall (form -> form) -> form.
type exists (form -> form) -> form.
type btm    form.

infix imp 8.
infix and 6.
```

```
infix or  4.

%% Subexponentials
type r index.
type l index.
type rules index.

type a form.
type b form.
type c form.
type d form.
type e form.
type f form.
type g form.
type h form.
type p form.
```

A.2.2 Código-fonte

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                             %
% Implementation of G1m's specification in LL using SELLF                     %
%                                                                             %
% Author: Giselle Machado N. Reis                                             %
%                                                                        2010 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

module g1m.
accumulate sellf.

g1m L R :- atomize_left L L1,
           atomize_right R R1,
           append L1 R1 F,
           prove_a ((k_elm r R1)::(k_elm l L1)::
                    (k_elm rules (
                        (sigma A\ sigma B\ ((my_not (bd (A imp B))) , ((bang l (? r (bu A))) , (? l (bd B))))::
                        (sigma A\ sigma B\ ((my_not (bu (A imp B))) , ((? l (bd A)) @ (? r (bu B))))::
                        (sigma A\ sigma B\ ((my_not (bd (A and B))) , ((? l (bd A)) ; (? l (bd B))))::
                        (sigma A\ sigma B\ ((my_not (bu (A and B))) , ((? r (bu A)) , (? r (bu B))))::
                        (sigma A\ sigma B\ ((my_not (bd (A or B))) , ((? l (bd A)) & (? l (bd B))))::
                        (sigma A\ sigma B\ ((my_not (bu (A or B))) , ((? r (bu A)) ; (? r (bu B))))::
                        (sigma A\ ((my_not (bd (forall A))) , (? l (bd (A X))))::
                        (sigma A\ ((my_not (bu (forall A))) , (pi x\ (? r (bu (A x)))))::
                        (sigma A\ ((my_not (bd (exists A))) , (pi x\ (? l (bd (A x)))))::
                        (sigma A\ ((my_not (bu (exists A))) , (? r (bu (A X))))::
                        (sigma B\ ((my_not (bd B)) , ((? l (bd B)) @ (? l (bd B))))::
                        (sigma B\ ((my_not (bd B)) , (bottom)))::
                        (sigma A\ ((my_not (bd A)) , (my_not (bu A))))::
                        (sigma A\ ((bang l (? r (bu A))) , (? l (bd A))))::
                        nil
                    ))::nil)
                    nil
                    F
                    0.


atomize_left nil nil.
atomize_left (H::L) ((bd H)::T) :- atomize_left L T.

atomize_right nil nil.
atomize_right (H::L) ((bu H)::T) :- atomize_right L T.

%% Polarity
```

```
neg (bd X).
neg (bu X).

%% Subexponentials
%% l and r are unrelated
pred l rules.
pred r rules.

cont rules.
weak rules.
```

A.3 Implementação de *MLJ*

A.3.1 Assinatura

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                   %
% Implementation of MLJ's specification in LL using SELLF          %
% Signature file.                                                   %
%                                                                   %
% Author: Giselle Machado N. Reis                                   %
%                                                          2010   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

sig mlj.
accum_sig sellf.

kind form type.

%% Predicates
type mlj (list form) -> (list form) -> o.
type atomize_left (list form) -> list o -> o.
type atomize_right list form -> list o -> o.

%% Brackets down and up
type bd form -> o.
type bu form -> o.

%% Connectives
type imp    form -> form -> form.
type and    form -> form -> form.
type or     form -> form -> form.
type forall (form -> form) -> form.
type exists (form -> form) -> form.
type btm    form.

infix imp 8.
infix and 6.
infix or  4.

%% Subexponentials
type r index.
type l index.
type rules index.

type a form.
type b form.
type c form.
type d form.
type e form.
type f form.
type g form.
```

```
type h form.
type p form.
```

A.3.2 Código-fonte

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                  %
% Implementation of MLJ's specification in LL using SELLF          %
%                                                                  %
% Author: Giselle Machado N. Reis                                  %
%                                                           2010   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

module mlj.
accumulate sellf.

mlj L R :- atomize_left L L1,
           atomize_right R R1,
           append L1 R1 F,
           prove_a ((k_elm r R1)::(k_elm l L1)::
                   ((k_elm rules (
                     (sigma A\ sigma B\ ((my_not (bu (A imp B))) , (bang l ((? l (bd A)) @ (? r (bu B))))))::
                     (sigma A\ sigma B\ ((my_not (bu (A and B))) , ((? r (bu A)) & (? r (bu B)))))::
                     (sigma A\ sigma B\ ((my_not (bu (A or B))) , ((? r (bu A)) @ (? r (bu B)))))::
                     (sigma A\ ((my_not (bu (forall A))) , (pi x\ (bang l (? r (bu (A x)))))))::
                     (sigma A\ ((my_not (bu (exists A))) , (? r (bu (A X)))))::
                     (sigma A\ sigma B\ ((my_not (bd (A imp B))) , ((? r (bu A)) & (? l (bd B)))))::
                     (sigma A\ sigma B\ ((my_not (bd (A and B))) , ((? l (bd A)) @ (? l (bd B)))))::
                     (sigma A\ sigma B\ ((my_not (bd (A or B))) , ((? l (bd A)) & (? l (bd B)))))::
                     (sigma A\ ((my_not (bd (forall A))) , (? l (bd (A X)))))::
                     (sigma A\ ((my_not (bd (exists A))) , (pi x\ (? l (bd (A x))))))::
                     ((my_not (bd btm)))::
                     (sigma A\ ((my_not (bd A)) , (my_not (bu A))))::
                     (sigma A\ ((bang l (bd A)) , (? r (bu A))))::
                   nil)))::nil)
                   nil
                   F
                   0.

atomize_left nil nil.
atomize_left (H::L) ((bd H)::T) :- atomize_left L T.

atomize_right nil nil.
atomize_right (H::L) ((bu H)::T) :- atomize_right L T.

%% Polarity
neg (bd X).
neg (bu X).

%% Subexponentials
cont r.
weak r.
cont l.
weak l.
cont rules.
weak rules.

pred r l.
pred l l.
pred l rules.
pred r rules.
```

A.4 Implementação de *LJQ**

A.4.1 Assinatura

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                              %
% Implementation of LJQ's specification in LL using SELLF                     %
% Signature file.                                                              %
%                                                                              %
% Author: Giselle Machado N. Reis                                              %
%                                                                        2010  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

sig ljq.
accum_sig sellf.

kind form type.

%% Predicates
type ljq (list form) -> (list form) -> o.
type atomize_left (list form) -> list o -> o.
type atomize_right list form -> list o -> o.

%% Brackets down and up
type bd form -> o.
type bu form -> o.

%% Connectives
type imp    form -> form -> form.
type and    form -> form -> form.
type or     form -> form -> form.
type btm    form.

infix imp 8.
infix and 6.
infix or 4.

%% Subexponentials
type r index.
type l index.
type f index.
type rules index.

type a form.
type b form.
type c form.
type d form.
type e form.
type p form.
```

A.4.2 Código-fonte

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                              %
% Implementation of LJQ's specification in LL using SELLF                     %
%                                                                              %
% Author: Giselle Machado N. Reis                                              %
%                                                                        2010  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

module ljq.
accumulate sellf.
```

```
ljq L R :-
    atomize_left L L1,
    atomize_right R R1,
    append L1 R1 F,
    prove_a ((k_elm l L1)::(k_elm r R1)::(k_elm f nil)::
             (k_elm rules (
                (sigma A\ ((my_not (bd A)) , (my_not (bu A)))):: 
                (((my_not (bd btm)) , top))::
                (sigma A\ sigma B\ ((my_not (bd (A imp B))) , ((bang l (? f (bu A))) , (bang r (? l (bd B))))))::
                (sigma A\ sigma B\ ((my_not (bu (A imp B))) , (bang l ((? l (bd A)) @ (? r (bu B)))))):: 
                (sigma A\ sigma B\ ((my_not (bd (A or B))) , ((bang r (? l (bd A))) , (bang r (? l (bd B))))))::
                (sigma A\ sigma B\ ((my_not (bu (A or B))) , (bang r ((? r (bu A)) @ (? r (bu B)))))):: 
                (sigma A\ sigma B\ ((my_not (bd (A and B))) , (bang r ((? l (bd A)) @ (? l (bd B)))))):: 
                (sigma A\ sigma B\ ((my_not (bu (A and B))) , ((bang r (? f (bu A))) , (bang r (? f (bu B))))))::
                nil)
             )::nil
            )
            nil
            F
            0.

atomize_left nil nil.
atomize_left (H::L) ((bd H)::T) :- atomize_left L T.

atomize_right nil nil.
atomize_right (H::L) ((bu H)::T) :- atomize_right L T.

%% Polarity

neg (bd X).
neg (bu X).

%% Subexponentials
cont r.
weak r.
cont l.
weak l.
cont rules.
weak rules.

pred r l.
pred l rules.
```

A.5 Implementação de *BAG*

A.5.1 Assinatura

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                   %
% Implementation of BAG's specification in LL using SELLF           %
% Signature file.                                                   %
%                                                                   %
% Author: Giselle Machado N. Reis                                   %
%                                                          2010     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

sig bag2.
accum_sig sellf.

kind prog type.

%% Predicates
```

```
type bag prog -> o.
type cmd prog -> o.

%% Commands

type load           T -> index -> prog -> prog.
type unload         index -> (T -> prog) -> prog.
type unload2        index -> (T -> T -> prog) -> prog.
type while          o -> (prog -> prog) -> prog -> prog.
type loop           index -> (T -> prog -> prog) -> prog -> prog.
type loop2          index -> (T -> T -> prog -> prog) -> prog -> prog.
type alt            prog -> prog -> prog.
type if             o -> (prog -> prog) -> (prog -> prog) -> prog -> prog.
type if_is_empty    index -> (prog -> prog) -> (prog -> prog) -> prog -> prog.
type new            index -> prog -> prog.
type done           prog.

infixl alt 9.

%% Subexponentials
type def index.
type i1 index.
type i2 index.
type i3 index.
type i4 index.

type l1 index.
type l2 index.
type l3 index.
type l4 index.
type visited index.
type edges index.
```

A.5.2 Código-fonte

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                               %
% Implementation of BAG's specification in LL using SELLF                      %
%                                                                               %
% Author: Giselle Machado N. Reis                                               %
%                                                                        2010   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

module bag2.
accumulate sellf.

bag P :-
    prove_a ((k_elm def (
                (sigma T\ sigma L\ sigma Prog\
                    (my_not (cmd (load T L Prog)) , ((? L T) @ ((cmd Prog) , true)))) ::
                (sigma L\ sigma Prog\
                    (my_not (cmd (unload L Prog)) ,
                        (sigma T\ (at L (my_not T) , ((cmd (Prog T)) @ bottom))))) ::
                (sigma L\ sigma Prog\
                    (my_not (cmd (unload2 L Prog)) ,
                        (sigma X\ sigma Y\ (at L (my_not [X,Y]) , ((cmd (Prog X Y)) @ bottom))))) ::
                (sigma C\ sigma Prog1\ sigma Prog2\
                    (my_not (cmd (while C Prog1 Prog2)) ,
                        ((C , ((cmd (Prog1 (while C Prog1 Prog2))) @ bottom)) ;
                            (not C , ((cmd Prog2) @ bottom))))) ::
                (sigma I\ sigma Kprog\ sigma Prog\
                    (my_not (cmd (loop I Kprog Prog)) ,
                        ( (sigma X\ (at I (my_not X) , (cmd (Kprog X (loop I Kprog Prog))))) ;
                            bang_empty I (cmd Prog) ))) ::
```

```
                (sigma I\ sigma Kprog\ sigma Prog\
                    (my_not (cmd (loop2 I Kprog Prog)) ,
                        ( (sigma X\ sigma Y\ ((at I (my_not [X,Y]) ,
                            (cmd (Kprog X Y (loop2 I Kprog Prog)))))) ; bang_empty I (cmd Prog) ))::
                (sigma Prog1\ sigma Prog2\
                    (my_not (cmd (Prog1 alt Prog2)) , ((cmd Prog1) ; (cmd Prog2))))::
                (sigma C\ sigma Prog1\ sigma Prog2\ sigma Cont\
                    (my_not (cmd (if C Prog1 Prog2 Cont)) ,
                        ((C , ((cmd (Prog1 Cont)) @ bottom)) ; (not C , ((cmd (Prog2 Cont)) @ bottom)))))::
                (sigma L\ sigma Prog1\ sigma Prog2\ sigma Cont\
                    (my_not (cmd (if_is_empty L Prog1 Prog2 Cont)) ,
                        (bang_empty L (cmd (Prog1 Cont)) ; (sigma X\ (at L X) @
                            ((cmd (Prog2 Cont)) @ bottom)))))::
                (sigma L\ sigma Prog\
                    (my_not (cmd (new L Prog)) , (create L (cmd Prog))))::
                    (my_not (cmd done) , top)::
                nil
            ))::(k_elm l1 ([1,4]::[2,3]::[3,2]::[4,1]::nil))::(k_elm l2 nil)::(k_elm l3 nil)::nil)
            nil
            (cmd(P)::nil)
            0.

%% Polarity
neg (cmd A).

%% Subexponentials
cont def.
weak def.

%%%% Subexponential indexes

weak i1.
weak i2.
weak i3.
weak i4.

cont i3.
cont i4.

pred l1 l2.
pred l2 l3.
pred i1 i2.
pred i2 i3.
pred i3 i4.

pred i1 i1.
pred i1 i3.
pred i1 i4.
pred i2 i2.
pred i2 i4.
pred i3 i3.
pred i4 i4.
pred l1 l1.
pred l1 l3.
pred l2 l2.
pred l3 l3.
```

Referências Bibliográficas

[Andreoli, 1992] Andreoli, J.-M. (1992). Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297--347.

[Andreoli & Pareschi, 1991] Andreoli, J. M. & Pareschi, R. (1991). Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9(3-4):445--473.

[Dyckhoff & Lengrand, 2006] Dyckhoff, R. & Lengrand, S. (2006). LJQ: a strongly focused calculus for intuitionistic logic. In Beckmann, A. & *et al.*, editores, *Computability in Europe 2006*, volume 3988 of *LNCS*, pp. 173--185. Springer.

[Gentzen, 1969] Gentzen, G. (1969). Investigations into logical deductions. In Szabo, M. E., editor, *The Collected Papers of Gerhard Gentzen*, pp. 68--131. North-Holland, Amsterdam.

[Girard, 1987] Girard, J.-Y. (1987). Linear logic. *Theoretical Computer Science*, 50:1--102.

[Girard, 1991] Girard, J.-Y. (1991). On the unity of logic. Technical Report 26, Université Paris VII.

[Hodas & Miller, 1994] Hodas, J. & Miller, D. (1994). Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327--365.

[Maehara, 1954] Maehara, S. (1954). Eine darstellung der intuitionistischen logik in der klassischen. *Nagoya Mathematical Journal*, pp. 45--64.

[Miller, 1989] Miller, D. (1989). Lexical scoping as universal quantification. In *Sixth International Logic Programming Conference*, pp. 268--283, Lisbon, Portugal. MIT Press.

[Miller, 1994] Miller, D. (1994). A multiple-conclusion meta-logic. In Abramsky, S., editor, *Ninth Annual Symposium on Logic in Computer Science*, pp. 272--281, Paris. IEEE Computer Society Press.

[Miller, 1999] Miller, D. (1999). Sequent calculus and the specification of computation. In Berger, U. & Schwichtenberg, H., editores, *Computational Logic*, volume 165 of *Nato ASI Series*, pp. 399--444. Springer.

[Miller & Pimentel, 2002] Miller, D. & Pimentel, E. (2002). Using linear logic to reason about sequent systems. In Egly, U. & Fermüller, C. G., editores, *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, volume 2381 of *LNCS*, pp. 2--23. Springer.

[Nadathur & Miller, 1988] Nadathur, G. & Miller, D. (1988). An Overview of λProlog. In *Fifth International Logic Programming Conference*, pp. 810--827, Seattle. MIT Press.

[Nigam, 2009] Nigam, V. (2009). *Exploiting non-canonicity in the sequent calculus.* PhD thesis, Ecole Polytechnique.

[Nigam et al., 2010] Nigam, V.; Pimentel, E. & Reis, G. (2010). Specifying proof systems in linear logic with subexponentials. In *Logical and Semantic Frameworks with Applications - LSFA.*

[Pimentel & Miller, 2005] Pimentel, E. & Miller, D. (2005). On the specification of sequent systems. In *LPAR 2005: 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, number 3835 in LNAI, pp. 352--366.

[Reis & Pimentel, 2010] Reis, G. & Pimentel, E. (2010). Using linear logic with subexponentials to implement logic interpreters. In *Proceedings of SBMF.*

[Troelstra & Schwichtenberg, 1996] Troelstra, A. S. & Schwichtenberg, H. (1996). *Basic Proof Theory.* Cambridge University Press.