SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura:

Sobre o uso da Gramática de Dependência Extensível na Geração de Língua Natural: questões de generalidade, instanciabilidade e complexidade

Jorge Marques Pelizzoni

Orientadora: ***Profa. Dra. Maria das Graças Volpe Nunes***

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Doutor em Ciências - Ciências de Computação e Matemática Computacional.

USP – São Carlos
Agosto de 2008

Sobre o uso da Gramática de Dependência Extensível na Geração de Língua Natural: questões de generalidade, instanciabilidade e complexidade

Jorge Marques Pelizzoni

A minha mãe e minhas irmãs. Todas ao mesmo tempo porque espero não escrever outra tese de doutorado nesta vida.

AGRADECIMENTOS

A Deus.

À grande pessoa de Maria das Graças Volpe Nunes, que, entre outros, é minha orientadora neste projeto. A ela especialmente pelo exemplo, tolerância e amparo.

A Claire Gardent e Denys Duchier, orientadores estrangeiros. A eles especialmente pela boa semente.

A Ralph Debusmann, que só conheço de *e-mail*, mas é como se fosse pessoalmente. A ele especialmente por um trabalho excelente, a colaboração e a simpatia.

À minha família, que hoje inclui o Thiago, a Beth, a Lina, o Belha, a Neneza, a Vacucha e a Chuzy. Ao primeiro especialmente pela coragem; e às três últimas pelo antídoto, muitas vezes ministrado em longas sessões de terapia de gatos energéticos.

Aos amigos. A todos eles especialmente pelo elo com a sanidade e pelo amor, às vezes à distância, mas sempre correspondido. Não os cito apenas por questões de ordem, pois cabem com folga em poucas linhas. Vocês sabem quem são.

À CAPES, CNPq, NILC, ICMC e entes familiares, pelo apoio financeiro.

A você.

Resumo

A Geração de Língua Natural (GLN) ocupa-se de atribuir forma lingüística a dados em representação não-lingüística (Reiter & Dale, 2000); a Realização Lingüística (RL), por sua vez, reúne as subtarefas da GLN estritamente dependentes das especificidades da língua-alvo. Este trabalho objetiva a investigação em RL, uma de cujas aplicações mais proeminentes é a construção de módulos geradores de língua-alvo na tradução automática baseada em transferência semântica. Partimos da identificação de três requisitos fundamentais para modelos de RL – quais sejam generalidade, instanciabilidade e complexidade – e da tensão entre esses requisitos no estado da arte. Argumentamos pela relevância da avaliação formal dos modelos da literatura contra esses critérios e focalizamos em modelos baseados em restrições (Schulte, 2002) como promissores para reconciliar os três requisitos. Nesta classe de modelos, identificamos o recente modelo de Debusmann (2006) – *Extensible Dependency Grammar* (XDG) – e sua implementação — o *XDG Development Toolkit* (XDK) — como uma plataforma especialmente promissora para o desenvolvimento em RL, apesar de jamais utilizada para tal. Nossas contribuições práticas se resumem ao esforço de tornar o XDK mais eficiente e uma formulação da disjunção inerente à lexicalização adequada à XDG, demonstrando suas potenciais vantagens numa sistema de GLN mais completo.

ABSTRACT

Natural Language Generation (NLG) concerns assigning linguistic form to data in non-linguistic representation (Reiter & Dale, 2000); Linguistic Realization (LR), in turn, comprises all strictly target language-dependent NLG tasks. This work looks into RL systems from the perspective of three fundamental requirements — namely generality, instantiability, and complexity — and the tension between them in the state of the art. We argue for the formal evaluation of models against these criteria and focus on constraint-based models (Schulte, 2002) as tools to reconcile them. In this class of models we identify the recent development of Debusmann (2006) — Extensible Dependency Grammar (XDG) — and its implementation — the XDG Development Toolkit (XDK) — as an especially promising platform for RL work, in spite of never having been used as such. Our practical contributions comprehend a successful effort to make the XDK more efficient and a formulation of lexicalization disjunction suitable to XDG, illustrating its potential advantages in a full-fledged NLG system.

LISTA DE FIGURAS

Figura 1: listagem em Oz demonstrando a colaboração entre dois propagadores (números de linha incluídos para facilitar a explanação) 19

Figura 2: dependências entre os parâmetros do Manati. Uma seta de *X* para *Y* indica que *X* é definido em termos de *Y* 43

Figura 3: projeções de um grafo *g* (omitidas as relações das arestas). Se *a*, *b* e *c* constituem todas as dimensões de *g*, trata-se da visão do grafo como a soma de suas projeções 52

Figura 4: grafo hexadimensional componente de uma análise da frase "Peter promises Mary to smile." (com anotação prosódica de "Peter" e "smile" na sentença dada) segundo a gramática `diss.ul`, distribuída com o XDK 1.7. Diretamente à esquerda de cada projeção posiciona-se o identificador (ID, IS, LP, PA, PS ou SC) da sua respectiva dimensão 54

Figura 5: projeções em SC alternativas para as duas análises de "Every man loves a woman." segundo `diss.ul`. SC_1 e SC_2 correspondem respectivamente às equações **(IV.1)** e **(IV.2)**. As projeções nas demais dimensões (incluindo PA, aqui r7epresentada) ficam constantes. 56

Figura 6: projeções em ID e LP (as duas projeções sintáticas da gramática `diss.ul`) para uma análise de "*With whom does Peter laugh?*" 58

Figura 7: declaração da dimensão ID 63

Figura 8: definição dos rótulos das arestas em ID 64

Figura 9: definição dos atributos não-lexicais de ID e tipos auxiliares 64

Figura 10: definição dos atributos lexicais em ID. Utilizam-se aqui os tipos auxiliares definidos na Figura 8 e na Figura 9. 65

Figura 11: manifesto de que ID é, em geral, uma projeção não-vazia 65

Figura 12: aplicação de princípios em ID 68

Figura 13: princípios multidimensionais em IDPA 69

Figura 14: definição mínima da dimensão LEX 70

Figura 15: o léxico numa gramática XDG com três dimensões *a*, *b* e *c*. A área hachurada corresponde a uma única entrada lexical, composta pela simples agregação dos atributos lexicais de todas as dimensões 70

Figura 16: parte de uma entrada lexical para "*promises*" compatível com os dados lexicais de `diss.ul` 71

Figura 17: uso de classes lexicais num excerto (descontínuo) de `diss.ul` relativo à definição de "*I*" e "*her*" 72

Figura 18: disjunção e conjunção em classes lexicais num excerto (descontínuo) de `diss.ul` relativo à definição de "*promises*" 74

Figura 19: definição mais eficiente da classe `id_non3sg`, alternativa à presente na Figura 18 76

Figura 20: a satisfação lexical de uma análise XDG implica um mapeamento de vértices em entradas lexicais, aqui representado pelo padrão de preenchimento dessas entidades .. 77

Figura 21: conjunto $\{e_1, e_2, e_3\}$ de possíveis entradas lexicais para um vértice *v* e algumas correlações entre valores de atributo num dado instante da geração de uma análise XDG 79

Figura 22: versão em PL do princípio `agreement`, extraído da distribuição do XDK 81

Figura 23: cerne de uma definição nativa do princípio `agreement`, equivalente ao fonte PL da Figura 29 (pág. 111) 84

Figura 24: DFD de nível 0 do PW destacando o ponto de inserção do QO 87

LISTA DE TABELAS

Tabela 2: exemplos de expressões compostas de caminho em grafos e seus respectivos significados 51

Tabela 3: principais constructos da PL e seus respectivos significados. A letra ϕ representa subexpressões lógicas; ε, subexpressões não-lógicas; τ, expressões de tipo; e as demais, identificadores de variável dos seguintes tipos: dimensão (*d*), rótulo de aresta (*l*), vértice (*u* e *v*) e qualquer (*x* e *y*) 80

Tabela 4: variáveis-conjunto de vértices, ditas gráficas, que definem as projeções de uma análise XDG segundo o XDK 91

Tabela 5: regras básicas de cálculo de $\{\phi\}_x$, a eliminação da variável *x* da fórmula ϕ 96

Tabela 6: regras gerais de aprofundamento de quantificadores. 98

Tabela 7: regras de aprofundamento específicas para quantificadores $Q \in \{\forall, \exists\}$ 98

Tabela 8: regras de aprofundamento específicas para os quantificadores $Q \in \{\forall!, \exists!\}$. 99

Tabela 9: regras de cálculo de $\{\phi\}_x$ para expressões envolvendo quantificadores 100

Tabela 10: regras de alçamento de expressões livres 102

SUMÁRIO

I Introdução ... 1

Domínio: Realização Lingüística ... 1

Generalidade, instanciabilidade e complexidade ... 2

Metodologia: analítica ... 4

Requisitos de interesse ... 5

Lacuna ... 9

Meios e fins ... 10

Tese ... 11

Estratégia de Defesa ... 12

Estrutura ... 13

Índice remissivo ... 13

II Programação Concorrente por Restrições ... 15

Um modelo computacional concorrente para a propagação de restrições ... 18

Colaboração e sinergia ... 22

Completude e distribuição ... 22

Criação de modelo ... 26

III Modelos por Restrições para a Geração de Língua Natural ... 27

III.1 Geração de expressões referenciais ... 28

III.2 Agregação ... 30

III.3 Lexicalização ... 34

Gardent &Thater (2001) – geração com uma gramática baseada em descrições arbóreas ... 35

Melhorando a propagação: Gramáticas de Dependências ... 38

Koller & Striegnitz (2002) – geração como análise sintática ... 40

Pelizzoni & Nunes (2005) – Deconversão UNL por meio de programação multiparadigmática 41
IV Introdução à XDG 49
IV.1 Análises XDG 50
Grafos 50
Atributos 58
IV.2 Gramáticas XDG 62
Tipo das análises 62
Princípios gramaticais 66
Léxico 69
Satisfação e sincronização lexical 76
IV.3 Principle Writer 79
IV.4 Considerações finais 81
V Tornando o *Principle Writer* uma Realidade Prática 83
V.1 Otimizações 86
V.1.1 Eliminação de quantificadores 87
Uma fonte de complexidade 87
Restrições sobre conjuntos finitos 88
Restrições reificadas 89
Uma oportunidade 91
EQ básica 93
EQ básica: aprofundamento de quantificadores 97
EQ básica: quantificadores aninhados 99
EQ básica: fórmulas livres 101
EQ básica: ajustes superficiais 104
EQ de seleção 105
EQ de seleção: expressões de pertinência 106

EQ de seleção: *oneOrMore* e *all* 109
V.1.2 Eliminação de subexpressões comuns 113
V.2 Avaliação 117
VI Capturando a Disjunção na Lexicalização com a XDG 121
VI.1 Modelagem 122
A entrada da Realização Lingüística 122
Deleção como substância da disjunção 123
Como os vértices se relacionam 125
Referentes, argumentos e assim os vértices se encontram 125
Completude e composicionalidade semântica 127
Classes de co-referência, concentradores e revisão de PA e DS 128
Combatendo a sobre-redundância 132
Restrições redundantes de composicionalidade 134
VI.2 Desempenho e considerações finais 137
VII Considerações Finais e Trabalhos Futuros 138
Referências bibliográficas 139
Índice Remissivo 147

I Introdução

Domínio: Realização Lingüística

Informalmente, a Geração de Língua Natural (GLN) corresponde à produção automática de texto, falado ou escrito. Reiter & Dale (2000) muito propriamente a resumem como a geração de representação lingüística para dados em representação não-lingüística. Módulos de GLN usam algum tipo de conhecimento sobre a língua-alvo e o domínio da aplicação para gerar relatórios, mensagens de ajuda, resumos, falas em diálogos, perguntas, traduções, etc. Como estes mesmos autores apontam, trata-se de uma atividade potencialmente complexa, envolvendo processamento tanto lingüístico (e.g. lexicalização, agregação e geração de expressões referenciais) como extralingüístico (e.g. determinação de objetivos comunicativos, seleção de conteúdo e diagramação). A Realização Lingüística (RL), por sua vez, define-se aqui como todo o subconjunto lingüístico da GLN, ou seja, como a parte do processamento que, em algum grau, dependa necessariamente das especificidades da língua-alvo para ser definida.

Ambos os problemas ainda estão bastante longe de terem sido modelados satisfatoriamente, porque subsumem diversas questões não respondidas pela Inteligência Artificial, a Lingüística Computacional, a Lingüística e a Filosofia, entre outros ramos da empresa científica. Mesmo em humanos, boas habilidades de redação são ainda um diferencial e reconhecidas como decorrentes do que vulgarmente se chama "inteligência" e que parece agrupar todas as capacidades humanas que resistem à reprodução por computador, ou melhor, a uma formulação algorítmica. Complementarmente, mesmo os níveis mais ordinários de competência lingüística têm se provado difíceis de modelar. Por exemplo, operações tão rotineiras e triviais entre humanos como a resolução de anáfora e desambiguação lexical não dispõem ainda de modelos satisfatórios.

Grosso modo e muito lugar-comumente, o objetivo deste trabalho é buscar um modelo melhor para um problema ainda em aberto, qual seja a RL. O termo "modelo" é empregado aqui mui-

to como sinônimo de “solução geral” e foi preferido a este último termo por implicar mais claramente a idéia de *aproximação*, fundamental em Computação. Boa parte deste capítulo introdutório será dedicada a introduzir conceitos e desenvolver uma argumentação que permitam apreciar como nosso modelo pode ser melhor e por que isso interessa, ou seja, que nos permitam explicitar a lacuna no estado da arte com a qual estamos lidando. Em última análise, essa lacuna consiste numa tensão entre os três requisitos primordiais – axiomáticos mesmo – que valem para qualquer modelo computacional, a saber: generalidade, instanciabilidade e complexidade. É exatamente por dar boi a esses nomes que iniciamos nossa argumentação rumo à justificação e delimitação de nosso objeto de pesquisa.

Generalidade, instanciabilidade e complexidade

Ter a generalidade o estatuto de troféu é certamente um legado da Matemática à teoria e prática em Computação. De fato, o corrente sucesso de conceitos como os de *framework*, *template*, *design pattern* e mesmo de algo tão rotineiro quanto a noção de parâmetro deve-se, em última análise, à alta cotação da generalidade nesse meio, a qual se justifica mais por razões econômicas do que estéticas ou hereditárias, já que dela decorre a reusabilidade e, por conseguinte, menores custos. No entanto, sempre se contrapõe à generalidade uma questão que, de tão onipresente, raramente é sequer mencionada, a saber: *instanciabilidade*.

Um modelo é *instanciável* para um problema específico se é *factível* fixar os seus parâmetros de forma a aproximar satisfatoriamente a solução desejada. Em termos absolutos/booleanos, trata-se de um conceito praticamente intrínseco à noção de modelo e, por conseguinte, pouco interessante *per se*. Entretanto, relativisticamente, torna-se uma questão de ordem e ubíqua. Na resolução de problemas reais, sempre importa determinar qual entre diversos modelos disponíveis tem o menor custo de aplicação e ainda se este mínimo é suficiente, o que pode motivar a busca por novos modelos. Instanciabilidade é o aspecto avaliado nessas ocasiões, o qual em grande parte corresponde a *custo* de instanciação/aplicação.

Vale observar que a instanciabilidade é uma questão exclusivamente de tempo de projeto. Sua contraparte em tempo de execução é a bem-conhecida *complexidade (computacional)*. Mesmo que, num dado caso, um modelo seja suficientemente geral e tenha sido devidamente instanciado, é perfeitamente possível que haja elementos do conjunto das possíveis entradas cujo processamento, apesar de teoricamente possível, exceda as contingências computacionais (tempo e espaço) vigentes. Como parte da complexidade de uma instância é muitas vezes intrínseca ao modelo subjacente, esse é outro fator preponderante na escolha de modelos.

Não é exagero dizer que a tensão entre generalidade, instanciabilidade e complexidade – entre o possível e o factível – move a Computação. A generalidade se refere aos limites teóricos de um modelo, ou seja, restringe o conjunto dos problemas a que este se aplica ou, equivalentemente, define quão aproximadas são as soluções obteníveis. Conceitos diretamente correlatos são *cobertura* (de fenômenos) e *expressabilidade*. A instanciabilidade, por sua vez, relaciona-se com a *expressividade* de um modelo e, juntamente com a complexidade, determina quanto o modelo será instanciado na prática ou quão útil ele é tal como está. Por exemplo, a Máquina de Turing *pode* ser considerada um modelo de máxima generalidade, mínima instanciabilidade e complexidade amplamente variável, dependente da instanciação. Pensar no que lhe falta para constituir um modelo computacional prático é um bom exercício para obter uma lista de desejos decorrentes do impulso primordial de instanciabilidade. Só para citar alguns: abstração, modularidade, desacoplamento, encapsulamento, concisão, manutenibilidade, reusabilidade[1], gerenciabilidade, escalabilidade, prototipabilidade. Daí já se pode pressentir a ascendência da instanciabilidade sobre toda a Engenharia de Software.

[1] Reusabilidade é o sinal mais legítimo de equilíbrio entre generalidade e instanciabilidade.

Quando ainda não existe modelo satisfatório para um problema, o que necessariamente ocorre é que os modelos conhecidos apresentam deficiência em pelo menos um dos fatores do trinômio – ou muitas vezes em todos. Baixa generalidade acarretará aproximações grosseiras demais, pois o problema em questão, na verdade, não estará sendo resolvido/modelado em toda a sua generalidade, havendo cobertura de casos isolados apenas. Baixa instanciabilidade/alta complexidade será caracterizada por demanda impraticável em recursos (financeiros, temporais, cognitivos, computacionais etc.) em tempo de projeto/execução. Por outro lado, um modelo satisfatório apresentará níveis aceitáveis de generalidade, instanciabilidade e complexidade. Acima de tudo, é exatamente nesses três quesitos que dois ou mais modelos aplicáveis concorrerão quando da resolução de um dado problema. Em conseqüência, a pesquisa de novos modelos procurará sempre por um equilíbrio ainda não atingido pelo trinômio.

Metodologia: analítica

Propor um modelo melhor implica necessariamente compará-lo com os já disponíveis, ou seja, um concurso. Diferentemente do que ocorre em outras áreas mais afortunadas, o nosso concurso de modelos de RL não deverá ser empírico. Em primeiro lugar, não há sequer um acordo sobre a entrada dos modelos, que varia de simples tabelas de dados (Eddy et al., 2001) a hipóteses de representação conceitual (Martins, 2004; Martins et al., 2002; Schank & Rieger, 1974; Lytinen, 1992) mais ou menos complexas, mais ou menos defensáveis, mais ou menos especializadas, mas sempre incompletas – ou seja, comprometendo a generalidade. Em específico, ou não há isomorfismo entre as linguagens de entrada ou os modelos estão instanciados para domínios distintos, o que torna qualquer tentativa de *benchmarking* impossível ou, no mínimo, pouco significativa. Outro ponto que desaconselha uma abordagem empírica neste caso é que uma tal abordagem estaria comparando *instâncias* de modelos, não os modelos propriamente ditos. Em outras palavras, (i) a figura do instanciador (seja humano ou computacional), (ii) sua competência de instanciação e (iii) o favorecimento de um ou outro modelo por parte dos casos de teste surgem como variáveis difíceis de neutralizar experimentalmente.

Como alternativa mais produtiva num campo em que os modelos são, reconhecidamente, ainda tão primitivos e díspares, advogamos uma abordagem analítica, ou seja, com base em propriedades formais e de suas conseqüências, ou melhor, *limites* lógicos. Trata-se de uma rotina de levantar requisitos decorrentes das premissas básicas de generalidade, instanciabilidade e complexidade e, para cada requisito *R* e modelo *M*, (i) verificar que simplesmente não há suporte para *R* em *M* ou, caso contrário, (ii) analisar como *M* contempla *R*, julgando sua instanciabilidade/complexidade para este requisito.

Naturalmente, levantar e detalhar *todos* os requisitos de generalidade da RL – para não mencionar discuti-los para cada modelo disponível – já seria tarefa ambiciosa demais e, portanto, fora do escopo desta tese. Cabe identificar um conjunto de requisitos de alto nível que seja a um só tempo suficientemente representativo e *problemático*, na medida em que nem todos sejam satisfeitos nos modelos atuais.

Requisitos de interesse

Partindo (i) do trinômio generalidade-instanciabilidade-complexidade, (ii) do subconjunto lingüístico das tarefas de geração identificadas por Reiter & Dale (2000) e (iii) da análise dos pontos fortes e fracos de alguns modelos de geração da literatura (Robin & McKeown, 1996; Stone & Doran, 1997; Eddy et al., 2001; Eddy, 2002; Koller & Striegnitz, 2002; Williams, 2003; Kibble & Power, 2004; Pelizzoni & Nunes, 2005), derivamos o seguinte conjunto de requisitos de interesse:

a) **suporte à lexicalização:** é senso comum que uma mesma “mensagem” possa ser expressa lingüisticamente – *verbalizada* – de várias maneiras diferentes. Um modelo de RL suficientemente geral deve permitir a especificação de alternativas – sintáticas e lexicais – de verbalização para uma mesma entrada ou componentes da entrada. É interessante notar que a **lexicalização** – ou seja, a tarefa de optar por uma alternativa lexical ou sintática e fazê-la funcionar – é tudo o que alguns modelos de RL acabam por fazer;

b) otimalidade: num conjunto de verbalizações aparentemente equivalentes, geralmente umas são preferidas a outras, em função dos mais variados fatores, tais como gênero, registro, clareza, concisão, tempo/espaço disponível para emissão/apresentação, etc. De forma abstrata, pode-se dizer que (i) a ordem de preferência é estabelecida por algum tipo de *medida de qualidade*, cuja definição varia de acordo com a aplicação, e que (ii) *otimalidade* é o atributo característico dos modelos parametrizados para medidas de qualidade e equipados para, dada uma tal medida, gerar ótimos globais *de forma escalável*, ou seja, com complexidade aceitável;

c) modularidade: a instanciabilidade implica modularidade, o que, do ponto de vista macro de um modelo, significa que este deverá dispor de um conjunto de parâmetros a serem instanciados de forma tão ortogonal, desacoplada, estruturada, reusável, etc. quanto possível. Entenda-se aqui "parâmetro" como *qualquer* lacuna de especificação prevista pelo modelo, que estará instanciado ao se preencherem cada uma dessas lacunas, ou seja, *ao se fixarem seus parâmetros* e assim se obter um módulo de RL completo. Um parâmetro pode, por exemplo, representar diretamente uma tarefa (possivelmente sendo preenchido mesmo por um procedimento) ou ainda restrições/invariantes a serem satisfeitas pelas soluções procuradas, estratégias, heurísticas, léxico, formalismo gramatical/de entrada e assim por diante;

d) suporte à reusabilidade/abstração: os módulos se tornam tanto mais interessantes quanto mais reusáveis forem, o que se obtém por meio de abstração. Ou seja, é interessante que eles próprios também possam ser parametrizados e, portanto, aplicados diversas vezes dentro de uma mesma instanciação do modelo;

e) sinergia: se a instanciabilidade implica modularidade e parametrização, a otimalidade implica sinergia entre os parâmetros de um modelo ou os módulos de uma instância. De acordo com a literatura, a forma mais fácil de comprometer a sinergia e, portanto, a otimalidade é dispor os módulos/tarefas em *pipeline*, arquitetura muitas vezes preferida (Reiter

& Dale, 2000) por favorecer a instanciabilidade. Numa tal configuração, cada submódulo não pode senão encontrar uma solução ótima local para sua subtarefa, já que executa sem conhecimento (ou colaboração) dos próximos estágios. Infelizmente, a "soma" de vários ótimos locais muitas vezes não resulta num ótimo global, que, se requerido, só poderá ser encontrado com recurso a intenso *backtracking*, elevando a complexidade.

Como alternativa, existem as **arquiteturas integradas**, em que as diversas tarefas são e-xecutadas de forma entremeada, concorrente e, portanto, mais compatível com a noção de sinergia. O argumento normalmente usado contra esse tipo de arquitetura é que ele compromete a instanciabilidade, levando a implementações monolíticas. Em outras palavras, mesmo que ainda persista alguma partição em módulos, esta não guarda um mapeamento intuitivo com os modelos de análise de alto nível preferidos, que se expressam em termos de tarefas e descrições lingüísticas. Um dos pontos altos de nossa tese é exatamente fornecer evidências de que seja possível obter uma arquitetura integrada de RL que preserve tal mapeamento, ou seja, que tenha uma maior "continuidade de análise" e seja, portanto, mais instanciável;

f) **entrada como hipótese de representação conceitual**, do tipo que poderia, por exemplo, ser usado num projeto de tradução automática baseada em transferência semântica. Trata-se da entrada dos modelos de RL mais complexos e gerais, com espectro mais amplo de aplicação;

g) **entrada plurissentencial:** uma séria restrição apresentada por algumas linguagens de entrada (por exemplo, a UNL [Martins, 2004; Martins et al. 2002]) é a *monossentencialidade*, ou seja, a de que cada sentença de um texto a ser gerado seja representada independentemente, sem referência às demais. Sob essa restrição, o texto acaba sendo representado como uma seqüência de representações formalmente desconexas. A mais direta das implicações disso é que a elaboração da entrada – já organizada em sentenças e dispondo de elementos de coesão – já pressupõe processamento lingüístico, o que equiva-

le a dizer que *a entrada já é dependente de língua-alvo*. Por conseguinte, modelos de RL que assumem entrada monossentencial têm uma deficiência inata em generalidade, já que a RL, por definição, é o subconjunto lingüístico da GLN;

h) entrada sem estrutura textual fina: interessam-nos modelos cuja entrada, além de plurissentencial, seja a mais subespecificada possível quanto a estrutura textual[2]. Em especial, não deve haver demarcação de sentenças dada, o que se justifica pelos mesmos motivos apresentados em (g);

i) suporte à agregação: qualquer modelo de RL com entrada plurissentencial e sem estrutura textual fina deve necessariamente dar suporte à tarefa de *agregação* (Reiter & Dale, 2000), que pode ser resumida como a distribuição da informação a ser verbalizada em sentenças e parágrafos, bem como a sua ordenação, a geração de dispositivos coesivos e a manutenção da coerência;

j) entrada com biunivocidade entre descritores e referentes: algumas linguagens de entrada, especialmente as monossentenciais, já pressupõem cadeias de correferência pré-construídas para simplificar o modelo de geração correspondente. Para isso, permitem que um mesmo referente seja representado por descritores diferentes ao longo da entrada e requerem que o codificador desta crie descritores que espelhem mais diretamente as diferentes expressões (cor)referenciais esperadas na saída. Analogamente ao discutido em (g), explorar a não-biunivocidade entre descritores e referentes dessa forma é característi-

---

[2] Na acepção de Power et al. (2003). Estes autores advogam um plano particular de descrição lingüística paralelo aos mais usuais (sintático, semântico, retórico-discursivo, etc.) a que chamam **estrutura textual** (*text structure*). Uma tal estrutura é uma árvore ordenada cujos nós estão classificados segundo alguma hierarquia de composição, que pode ser, por exemplo:

seção > parágrafo > sentença textual > oração textual > sintagma textual.

co de formalismos de entrada dependentes de língua-alvo e de modelos deficientes em generalidade;

k) **suporte à geração de expressões referenciais:** qualquer modelo de RL que respeite a biunivocidade entre descritores e referentes deve dar suporte à geração de expressões referenciais de forma que favoreça – e sobretudo não comprometa – a textualidade da saída;

l) **extensibilidade:** é favorável à generalidade que um modelo tenha uma arquitetura aberta, permitindo adicionar outros módulos com impacto mínimo sobre a estrutura já existente. Por exemplo, tais módulos podem tratar de novos níveis de análise, como diagramação do texto e inclusão de figuras ou outros recursos comunicativos, ou ainda servir como *oráculos*, ou seja, módulos que respondem a consultas de outros módulos, recorrendo a bases de conhecimento, processamento inteligente ou ainda a um ajudante humano.

Lacuna

Em resumo, nossos requisitos podem ser expressos como (i) variedade de parâmetros (em decorrência de generalidade e extensibilidade), (ii) configuração amigável (instanciabilidade) e (iii) sinergia entre parâmetros ou eficiência na busca por uma solução (otimalidade, baixa complexidade). Reconciliar esses requisitos é o grande motivo de nossa tese e a lacuna que tenta preencher. Na literatura, encontramos ora (i) em detrimento de (ii), como em Stone & Doran (1997) [integração – pouco instanciável], ora vice-versa, como em Eddy (2002) [otimalidade] e Robin & McKeown (1996) [agregação e lexicalização – 1 sentença bem complexa, revisão – pipeline sem otimização], que reduzem consideravelmente o número de parâmetros tratados; e sempre (iii) em oposição a (i). Por exemplo, Kibble & Power (2004), num dos trabalhos recentes em GLN mais interessantes e que mais influenciaram nosso modelo, cobre elegantemente um espectro notável de parâmetros, mas reporta uma complexidade $\Omega(5^n)$, onde n é o número de proposições (lógicas) da entrada, o que limita sobremaneira a aplicabilidade do modelo.

Meios e fins

Nossa meta é exatamente avançar em direção a um modelo de RL que equacione a tensão vigente entre generalidade, instanciabilidade e complexidade. Nossa abordagem ao problema é inédita em três frentes, a saber:

- **foco:** não temos notícia de um trabalho que fraseie ou ataque o problema com a mesma ênfase nos três requisitos primários. É importante observar desde já que não pretendemos propor soluções completas para as tarefas de lexicalização, agregação ou geração de expressões referenciais, já que cada qual consiste num problema suficientemente complexo para merecer um trabalho independente. Nosso interesse é principalmente na capacidade de um modelo para suportá-las conjuntamente e atender aos demais requisitos. Em outras palavras, não objetivamos nem oferecemos um gerador completo para língua nenhuma, apenas procuramos um método de construção – ou arquitetura – de geradores com propriedades vantajosas;

- **tecnologia:** aplicamos intensivamente uma ferramenta ainda não explorada nos outros trabalhos correlatos – a Programação Concorrente por Restrições[3] (**PCR**, vide Van Roy & Haridi, 2004, e Schulte, 2002). Essa tecnologia possibilita podar drasticamente a busca, além de fornecer poder expressivo diferenciado, permitindo dissociar as semânticas declarativa e operacional de um programa, ou seja, especificar independentemente um modelo (aspecto declarativo) e uma estratégia de busca por uma solução (aspecto imperativo). Apresentamos uma breve introdução a esse paradigma de programação no Capítulo II;

- **metáfora unificadora:** elegemos como substrato conceitual para nosso modelo a Gramática de Dependência Extensível (**XDG**, do inglês *Extensible Dependency Grammar*), descrita em (Debusmann, 2006) e melhor formalizada em (Debusmann, 2007). Não se tra-

---

[3] Em inglês, *Concurrent Constraint Programming*.

ta propriamente de um formalismo gramatical, mas de uma linguagem de descrição de **grafos multidimensionais**, os quais podem ser utilizados para representar análises/descrições de objetos lingüísticos. Melhor dizendo, a XDG dá total liberdade e amplos dispositivos para desenvolver, num estilo estritamente declarativo, teorias do que sejam descrições bem-formadas dos objetos de interesse – sejam eles sentenças, textos ou outros – contanto que cada descrição consista num um conjunto de grafos[4] que compartilhem de um mesmo conjunto de vértice. Cada grafo descreve uma **dimensão** distinta do objeto analisado; e o número de grafos numa análise, seu significado e as relações lógicas que estabelecem entre si não estão fixados *a priori*, devendo ser especificados arbitrariamente por cada gramática XDG a partir de um repertório de primitivas. Portanto, a XDG permite modelar os diversos aspectos de um texto a ser gerado por meio de uma metáfora – a de dimensões paralelas e sincronizadas – e linguagem unificadoras. Além disso, ela já incorpora boa parte do que podemos desejar em termos de instanciabilidade e extensibilidade. Não por acaso, a XDG se presta a uma implementação bastante direta em PCR; em específico, temos disponível um excelente ambiente de experimentação com a XDG – o *XDG Development Kit* (**XDK**)[5] – que é fortemente baseado em PCR, foi utilizado em toda nossa parte experimental e hoje embute alguns dos resultados deste doutorado.

Tese

Apresentada a fundamentação acima, podemos formular a tese que defendemos neste doutorado, qual seja:

No domínio da RL, um maior equilíbrio entre os requisitos de gene-

---

[4] Estritamente falando, tais grafos são, na verdade, dígrafos com nós e arestas rotuladas, estas por átomos e aqueles por matrizes atributo-valor. Entretanto, manteremos o termo "grafo" do original por simplicidade.

[5] http://www.ps.uni-sb.de/~rade/mogul/publish/doc/debusmann-xdk

> ralidade, instanciabilidade e complexidade pode ser atingido por um modelo com arquitetura integrada, baseado na XDG e implementado por PCR.

Assumimos como premissa que o conjunto levantado de requisitos secundários (listagem à página 5) seja suficientemente representativo e relevante.

Estratégia de Defesa

Para evitar falsas expectativas o quanto antes, vale dizer que a tese acima foi nosso norte ao longo de toda a pesquisa e o será ao longo de todo o texto. Entretanto, não chegaremos a prová-la cabalmente em nenhum momento. Antes, daremos os primeiros passos em sua direção e procuraremos oferecer evidências de que tal seja uma direção digna de ser seguida, apesar dos inúmeros desafios e questões em aberto.

Esperamos que os resultados deste doutorado sejam compreendidos como estudos de caso nas questões de alto nível que queremos abordar. Não é nosso objetivo desenvolver um gerador de texto qualquer, até porque isso nós chegamos a fazer nesse ínterim (Pelizzoni & Nunes, 2005), resultado abordado marginalmente no Capítulo III. Trataremos em detalhe apenas dos dois resultados que mais se coadunam com a tese. O primeiro deles nem se insere especificamente na RL ou a GLN, mas consiste em um esforço para domar a complexidade em nome da instanciabilidade. Trata-se de um otimizador para o código-objeto do módulo *Principle Writer* (PW) do *XDK*. O PW é um compilador que trouxe instanciabilidade para a até então ingrata tarefa de escrita de princípios gramaticais. Entretanto, antes do nosso otimizador, o código gerado, apesar de correto, não raro apresentava um grau de complexidade que proibia sua aplicação prática. O outro resultado abordado concerne a uma tarefa muito específica da RL, consistindo na modelagem da lexicalização na XDG. Demonstramos, nesse estudo de caso, como é a rotina de trabalho com a XDG e os problemas e vantagens que dela podem advir.

Estrutura

Este documento prossegue com uma introdução à Programação Concorrente por Restrições (Capítulo II), a tecnologia cujo potencial queremos testar, e uma revisão de alguns modelos correlatos de GLN (III). Apresentamos, então, uma introdução à XDG e sua prática corrente (IV). Nos capítulos subseqüentes, detalhamos as contribuições específicas deste doutorado, descrevendo trabalho em otimização de código (V) e na modelagem da lexicalização (VI). Por fim, discutimos trabalhos futuros e concluímos (VII).

Índice remissivo

Ao longo do texto, muitos termos e expressões estarão em **negrito** por corresponderem a entradas do *índice remissivo* consideradas especialmente interessantes. Dessa forma, esperamos agilizar o uso desse índice, que avaliamos como um recurso bastante oportuno num trabalho abundante em definições (ora inéditas, ora pouco familiares), referências cruzadas e seções longas. Não por coincidência, portanto, muitos dos termos em negrito ocorrem em meio às suas respectivas definições, implícitas ou explícitas. Reciprocamente, no índice remissivo, os números de página em negrito correspondem a ocorrências especialmente interessantes, em quantidade e qualidade de informações, tratando-se freqüentemente de definições.

II Programação Concorrente por Restrições

Apresentamos aqui uma breve introdução à **Programação Concorrente por Restrições (P-CR)** como entendida, por exemplo, por Van Roy & Haridi (2004), Schulte (2002) e Duchier et al. (2002) e aqui por vezes referida como o **paradigma concorrente de restrições**. Esse paradigma de programação é tipicamente usado para resolver problemas de otimização de Pesquisa Operacional, tais como escalonamento (*scheduling* e *timetabling*) e configuração. Características comuns a esses problemas as quais recomendam modelagem por restrições são:

a) não são resolvidos satisfatoriamente por nenhum algoritmo conhecido de complexidade polinomial;

b) são passíveis de descrição finita, como um conjunto (finito) R de relações lógico-matemáticas relativamente simples – ditas **restrições** – que se estabelecem dentro de um conjunto (finito) V de variáveis (no sentido matemático do termo) tais que:

i. para toda variável $v \in V$, R contenha pelo menos uma **restrição básica**, isto é, do tipo $dom(v) \subseteq D$, onde D é um conjunto finito dado, e $dom(v)$ – dito o **domínio** de v – denota o conjunto (ainda não determinado) de valores aceitáveis para v;

ii. haja um subconjunto de variáveis de interesse $S \subseteq V$ tal que qualquer solução possa ser dada como uma atribuição de valores a essas variáveis, ou seja, como uma função $f : S \to \bigcup_{v \in S} dom_0(v)$, onde $dom_0(v)$ – dito o **domínio inicial** de v – denota a interseção dos Ds presentes nas restrições básicas sobre v dadas.

Uma tal descrição é dita um **modelo** do problema; e a atividade de formular modelos, **modelagem**;

c) a enumeração de todos os possíveis *candidatos* a solução tem *complexidade combinatória*, ou seja, exponencial no tamanho *n* da entrada. Condições bastante usuais e suficientes para que isso ocorra são as seguintes:

i. $|S|$ cresce pelo menos polinomialmente com o tamanho *n* da entrada, isto é, $|S| \in \Omega(n)$;

ii. para toda variável $v \in S$, $dom_0(v) \geq 1$, ou seja, não é trivial verificar que o conjunto-solução é vazio;

iii. existe $S' \subseteq S$ tal que $\forall v \in S' \left(|dom_0(v)| \geq 2 \right)$ e $|S'| \in \Omega(n)$, ou seja, o número de variáveis de solução trivial (domínio inicial unitário) não reduz o restante a uma constante.

Sob essas condições e considerando que é $\prod_{v \in S} |dom_0(v)|$ o número de atribuições distintas às variáveis de interesse em qualquer caso, já temos uma complexidade exponencial como demonstrado abaixo. Em primeiro lugar, podemos limitar esse produtório à seguinte potência:

$$\prod_{v \in S} |dom_0(v)| \geq \prod_{v \in S'} |dom_0(v)| \geq 2^{|S'|}. \qquad \text{(II.1)}$$

De $|S'| \in \Omega(n)$, temos, por definição, que:

$$\exists M > 0, x_0 > 0 \ \left(n > x_0 \rightarrow |n| \leq M |S'| \right). \qquad \text{(II.2)}$$

Assumindo sem perda de generalidade que $|S'| > 0$ e dividindo ambos os lados da inequação em (II.2) por *M*, temos:

$$n > x_0 \rightarrow \frac{n}{M} \leq |S'|. \qquad \text{(II.3)}$$

De (II.3), podemos escrever:

$$n > x_0 \rightarrow 2^{\frac{n}{M}} \leq 2^{|S'|}. \quad \text{(II.4)}$$

Por meio da igualdade $2^{\frac{n}{M}} = \left(2^{M^{-1}}\right)^n$ e fazendo $b = 2^{M^{-1}}$, temos:

$$n > x_0 \rightarrow b^n \leq 2^{|S'|}. \quad \text{(II.5)}$$

De (II.1) e (II.5), pode-se concluir que:

$$n > x_0 \rightarrow b^n \leq \prod_{v \in S} |dom_0(v)| \quad \text{(II.6)}$$

Fazendo $M' = 1$, a inequação em (II.5) pode ser reescrita da seguinte forma:

$$n > x_0 \rightarrow \left|b^n\right| \leq M' \left|\prod_{v \in S} |dom_0(v)|\right|. \quad \text{(II.7)}$$

Por definição, a expressão (II.7) equivale a:

$$\prod_{v \in S} |dom_0(v)| \in \Omega(b^n). \quad \text{(II.8)}$$

Como podemos garantir que $b > 1$, fica provada a complexidade exponencial da enumeração dos candidatos a solução se satisfeitas as condições (i) a (iii) acima;

d) por outro lado, o tamanho das descrições deve ter complexidade polinomial no tamanho da entrada, ou seja, é desejável que $|R| \in \mathrm{O}\left(P_1(n)\right)$ e $|V| \in \mathrm{O}\left(P_2(n)\right)$, para duas funções polinomiais $\{P_i(x)\}$ quaisquer.

As características (a) e (c) são típicas de problemas de busca, ou seja, para os quais a **resolução por busca** (Russel & Norvig, 1995; Rich & Knight, 1991) é a abordagem de resolução aplicável por enquanto. Trata-se de uma abordagem bastante geral que (i) parte da descrição

de um espaço de candidatos a solução (usualmente de cardinalidade exponencial na entrada), dos quais um subconjunto expressivo (idem) não é satisfatório, e (ii) percorre este espaço até encontrar uma solução ou aproximação satisfatória. Usualmente, o emprego de resolução por busca requer o emprego de dispositivos específicos à aplicação que domem a complexidade da busca, ou seja, que evitem o percurso de boa parte do espaço de busca e cheguem à solução em complexidade quase polinomial, na maioria dos casos. Quando as características (b) e (d) também estão disponíveis, a PCR surge como candidata a um tal dispositivo. Em outras palavras, enquanto (a) e (c) tornam esse paradigma desejável, (b) e (d) tornam-no aplicável.

Deve ficar claro, portanto, que a PCR é uma especialização da resolução por busca. Pode-se dizer que o seu diferencial seja explorar as propriedades singulares de uma certa forma de descrever o problema – características (b) e (d) – de modo que (i) a descrição do espaço de busca seja a própria descrição (R, V) do problema e (ii) o espaço de busca seja *(re)ativo*, permitindo que pelo menos parte das conseqüências lógicas do que é conhecido ou assumido durante a busca sejam automaticamente inferidas *para eliminar parte das alternativas necessariamente falidas*. Essa (re)ação da descrição do problema é denominada **propagação de restrições**, o primeiro conceito fundamental na PCR que ficará mais claro mediante a explanação do modelo computacional subjacente.

Um modelo computacional concorrente para a propagação de restrições

A propagação (de restrições) surge ao se "dar vida" às **restrições não-básicas** pertencentes a *R*, que são aquelas relacionando duas ou mais variáveis. Ou seja, para toda relação não básica $r[v_i] \in R$ (onde *r* é um predicado relacional; e $[v_i], 1 \le i \le n, n \ge 2$, o vetor de variáveis relacionadas nesta aplicação de *r*), além de valer sua **semântica declarativa** usual – denotada $semdecl(r)[v_i]$ – na medida em que garantidamente será detectada a inconsistência de $semdecl(r)[v_i]$ contra uma determinação total de $[v_i]$, atribui-se-lhe também uma **semântica operacional** – denotada $semoper(r)[v_i]$. À aplicação de $semoper(r)$ sobre um vetor de vari-

áveis $[v_i]$ corresponde a criação de um agente concorrente – dito **propagador** – assegurando a invariante $semdecl(r)[v_i]$, geralmente uma relação matemática simples, como $v_1 = v_2 + v_3 + v_4$. Entretanto, cabe a cada propagador $semoper(r)[v_i]$ não só tentar detectar a inconsistência de $semdecl(r)[v_i]$ antes de uma determinação total de $[v_i]$, mas também executar algum mecanismo de inferência limitada para *manter a consistência de* $semdecl(r)[v_i]$, (i) reagindo a novas restrições básicas publicadas para qualquer v_i, (ii) calculando conseqüências lógicas para os demais $v_j, j \neq i$, na forma de novas restrições básicas, (iii) publicando-as e assim (iv) refinando as restrições básicas vigentes.

Por sua vez, todos os propagadores compartilhando quaisquer v_j tomarão conhecimento das novas restrições e propagarão suas conseqüências lógicas sobre outras variáveis, iniciando uma reação em cadeia. Por vezes, um programa concorrente por restrições termina num único ciclo de propagação (possivelmente com muitos vaivéns até atingir a estabilidade), sem que nenhuma escolha não-determinística seja feita. Para ilustrar como isso ocorre, vale apresentar um exemplo simples extraído diretamente da documentação do sistema de programação Mozart (www.mozart-oz.org), que implementa a linguagem Oz.

```
1  declare X Y
2  X::0#9
3  Y::0#9
4  X + Y =: 9          % propagador P1
5  2*X + 4*Y =: 24     % propagador P2
6  {Browse [X Y]}      % imprime o resultado parcial
```

Figura 1: listagem em Oz demonstrando a colaboração entre dois propagadores (números de linha incluídos para facilitar a explanação)

Considere o programa em Oz listado na Figura 1, que corresponde à descrição de um problema muito simples com variáveis de interesse *X* e *Y*, declaradas na linha 1. Nas linhas 2 e 3, temos a imposição de duas restrições básicas iniciais, a saber $dom(X) \subseteq \{0..9\}$ e $dom(Y) \subseteq \{0..9\}$. Mesmo num exemplo tão pequeno, já podemos pressentir sua complexidade

potencialmente exponencial, já que o espaço de busca conta com $9^n = 9^2 = 81$ candidatos a solução, dos quais podemos adiantar que apenas um é satisfatório. Nas linhas 4 e 5, dois propagadores P_1 e P_2 são criados, com semânticas declarativas $X + Y = 9$ e $2X + 4Y = 24$, respectivamente. Note que, em Oz, o símbolo = é o operador de unificação, enquanto o símbolo **=:** (bem como qualquer operador relacional $R \in \{=:, \backslash=:, >:, >=:, <:, =<:\}$, cuja semântica declarativa é respectivamente $\{=, \neq, >, \geq, <, \leq\}$) é um **construtor de restrições**, criando um propagador para uma (in)equação qualquer envolvendo adição, subtração e multiplicação de variáveis/constantes inteiras.

Nessa situação inicial, P_1 não é capaz de inferir nada; P_2, por outro lado, consegue deduzir novas restrições básicas para ambas as variáveis, estreitando seus respectivos domínios para $dom(X) \subseteq \{0..8\}$ e $dom(Y) \subseteq \{2..6\}$. Isso é possível porque sua semântica operacional percebe que, para $Y \in \{1, 7, 8, 9\}$, não existe *X* tal que satisfaça sua semântica declarativa. De posse das novas restrições básicas e por meio de processamento análogo, P_1 está agora em posição de estreitar o domínio de *X*. Temos então:

$$dom(X) \subseteq \{3..7\} \text{ e } dom(Y) \subseteq \{2..6\}.$$

Agora, P_2 pode contribuir de novo, com os seguintes resultados:

$$dom(X) \subseteq \{4..6\} \text{ e } dom(Y) \subseteq \{3..4\}.$$

Mais uma vez, P_1 é ativado e estreita mais um pouco o domínio de *X*. Temos então:

$$dom(X) \subseteq \{5..6\} \text{ e } dom(Y) \subseteq \{3..4\}.$$

Por fim, P_2 arremata com os resultados finais:

$$dom(X) \subseteq \{6\} \text{ e } dom(Y) \subseteq \{3\}.$$

No exemplo apresentado, vale observar o seguinte:

a) não houve qualquer tipo de busca na resolução deste exemplo. Trata-se de um caso especial em que a propagação foi suficiente para encontrar uma solução deterministicamente;

b) uma propriedade importante dos propagadores é efetuar **inferência/comunicação multi-direcional**. Pode-se comparar cada variável a um duto de comunicação em *n-way broadcast* entre todos os *n* propagadores que a compartilham. Novas restrições básicas sobre uma variável corresponderiam a mensagens difundidas por um desses dutos. Ao receber uma nova mensagem, um propagador potencialmente *propagará* essa mesma mensagem por todos os seus demais dutos de comunicação, não literalmente, mas em versões transformadas que interessem aos seus "interlocutores";

c) apesar de ser implementada como uma computação concorrente, na qual a ordem de execução dos propagadores não é determinística, o resultado da propagação é determinístico. Isso é garantido pelo **caráter necessariamente monotônico do refinamento das restrições básicas**. Portanto, sempre que um propagador publica uma nova mensagem/restrição $dom(v) \subseteq D$, é como se a única mensagem $dom(v) \subseteq D_n$ que ocupasse anteriormente o duto fosse substituída atomicamente pelo resultado da conjunção entre as duas mensagens, ou seja, $dom(v) \subseteq D_{n+1} = (D_n \cap D)$;

d) cada sistema de PCR oferece um **repertório de tipos de variável e de propagadores predefinidos**. Por vezes, esse repertório pode ser estendido pelo usuário, como ocorre com o sistema Mozart, mas isso geralmente requer perícia. Não raro estão disponíveis num mesmo repertório propagadores equivalentes em semântica declarativa, mas alternativos em semântica operacional, envolvendo algoritmos de complexidade e poder de inferência diferentes. Os tipos de variável mais populares atualmente e para os quais a PCR tem sido aplicada com mais sucesso são (i) inteiros não-negativos e (ii) subconjuntos

finitos de $\mathbb{N}$. Ambos são suportados no Mozart, e seu uso em PLN será exemplificado no Capítulo III.

Para uma descrição formal e completa de um tal modelo computacional, consulte Van Roy & Haridi (2004) e Schulte (2002). Para uma implementação, experimente o Mozart.

Colaboração e sinergia

De baixo nível que aparente ser em virtude da simplicidade dos objetos primitivos manipulados (elementos e subconjuntos de $\mathbb{N}$ e simples relações matemáticas entre estes), o paradigma concorrente de restrições, aliado a outros como o funcional e o de objetos, permite definir abstrações poderosas e desenvolver um estilo declarativo de programação em que uma rede de comunicação intrincada e altamente estruturada é montada implicitamente a partir da especificação do problema a ser resolvido. Por essa rede, cujos vértices e arcos são respectivamente variáveis lógicas e propagadores, a certeza flui automaticamente de forma a minimizar o número de escolhas não-determinísticas na busca por uma solução. Trata-se de uma arquitetura de resolução (i) **intrinsecamente colaborativa**, já que as variáveis são verdadeiros *repositórios de resultados parciais*, que são compartilhados direta ou indiretamente por meio de propagação, seguramente produzindo efeitos à distância, e, portanto, (ii) **potencialmente sinérgica**, já que variáveis a serem maximizadas podem normalmente pertencer à rede, de forma que restrições globais/locais surtam efeitos locais/globais, por propagação, de forma transparente.

Completude e distribuição

Propagadores efetuam inferência multidirecional, mas **estritamente local**. Em especial, durante a propagação de restrições, não há uma entidade superior tentando derivar teoremas das semânticas declarativas de dois ou mais propagadores. Além disso, a complexidade da semântica operacional de um propagador é sempre um fator a ser considerado; e, por vezes, opta-se por algoritmos menos custosos que, contudo, não são capazes de realizar todas as inferências

locais possíveis. Como conseqüência desses dois fatores – **localidade de inferência** e **poder de inferência (local)** – propagação por si só não é uma técnica completa de resolução, levando usualmente à **estabilidade consistente**, estado em que nenhum propagador pode contribuir com qualquer nova restrição básica.

Como já observamos, a propagação não passa de um dispositivo para domar a complexidade na abordagem de busca. Para complementá-la e assim compor a PCR propriamente dita, existe a **distribuição**, que representa exatamente o elemento de busca ou **não-determinismo** no paradigma. Sempre que se atinge estabilidade consistente e nem todas as variáveis de interesse estão determinadas, é necessário inserir alguma instabilidade para reiniciar a propagação. Esse é o objetivo de qualquer passo de distribuição, que consiste em elaborar uma restrição artificial C – dita **de distribuição** – e bifurcar o processo de busca, assumindo ora C, ora $\neg C$, para garantir completude. Ao se intercalar recursivamente propagação e distribuição, tem-se um método de resolução completo e potencialmente eficiente.

Aplicar recursivamente apenas distribuição já é um método completo. Para isso, basta, a cada passo, selecionar alguma variável de interesse v ainda não determinada e algum valor possível v_0 segundo a restrição básica vigente $dom(v) \subseteq D$ e assumir ora $dom(v) \subseteq \{v_0\}$, ora $dom(v) \subseteq D - \{v_0\}$. Entretanto, esse método corresponde à enumeração simples e tem, como já demonstramos, complexidade exponencial. Mesmo dentro da PCR, se a propagação não for suficientemente forte, teremos então um caso degenerado que se aproxima da enumeração e de sua complexidade. A propagação é tanto mais forte quanto mais for capaz de eliminar alternativas inválidas e, assim, minimizar os passos de distribuição.

Dois fatores são fundamentais para maximizar – ou fortalecer – a propagação, a saber:

a) **a estratégia de distribuição adotada**, ou seja, o procedimento que, dada uma situação de estabilidade consistente, elabora a restrição de distribuição, usada para bifurcar o procedimento de busca e introduzir instabilidade. É sabido que, para uma mesma modelagem,

diferentes estratégias podem ter eficiência bastante díspar, determinando o sucesso ou fracasso da aplicação da PCR. Trata-se de uma escolha dependente de aplicação; e, por vezes, a melhor estratégia só é selecionada – ou desenvolvida – mediante experimentação. A estratégia de distribuição representa o elemento heurístico da PCR.

Algumas estratégias simples, mas de ampla utilização, dividem a tarefa em dois passos, a saber:

i. selecionar uma entre as possivelmente várias variáveis de interesse ainda não determinadas. Uma estratégia de seleção comum é denominada *first-fail* ("falhe primeiro") e consiste em selecionar a variável de domínio mais restrito até o momento como uma heurística para eliminar casos absurdos mais cedo. Outro exemplo interessante consiste em selecionar a variável compartilhada pelo maior número de propagadores, apostando na suposição de que, ao perturbar este plexo, a instabilidade gerada será máxima; e

ii. aplicar um predicado fixo sobre a variável selecionada, gerando a restrição de distribuição. Exemplos desses predicados podem ser:

$$\lambda v. \quad dom(v) \subseteq \left\{ \min\left(curdom(v) \right) \right\}$$

$$\lambda v. \quad dom(v) \subseteq \left\{ \max\left(curdom(v) \right) \right\}$$

$$\lambda v. \quad v \leq \left\lfloor \frac{\max\left(curdom(v) \right) - \min\left(curdom(v) \right)}{2} \right\rfloor$$

onde $curdom(v)$ denota o conjunto limitante superior D da restrição básica $dom(v) \subseteq D$ vigente sobre o domínio de X.

Além de um repertório de estratégias de distribuição predefinidas, alguns sistemas de P-CR, como o Mozart, dão total liberdade ao programador para a definição de novas estratégias.

b) a modelagem propriamente dita, já que há freqüentemente mais de uma forma de descrever um mesmo problema, umas mais bem-sucedidas que outras. Em específico, uma preocupação constante dentro da PCR é a **eliminação de simetrias**. Diz-se haver uma **simetria** quando, para dois subconjuntos disjuntos $S_1, S_2 \subseteq S$ de variáveis de interesse, existe uma bijeção $p: S_1 \to S_2$ (ou seja, uma *substituição* de variáveis) tal que, se a atribuição de variáveis f é solução, então também o é uma atribuição f' construída da seguinte forma:

$$f'(v) = \begin{cases} f\left(p(v)\right), & v \in S_1 \\ f\left(p^{-1}(v)\right), & v \in S_2 \\ f(v), & v \in S - \left(S_1 \cup S_2\right) \end{cases}.$$

Informalmente falando, uma simetria surge quando permutar os valores de certos blocos de variáveis de interesse leva a soluções equivalentes. Isso introduz um não-determinismo intrínseco ao modelo, o que deixa os propagadores sem ação e só poderá ser resolvido por distribuição. Veja um exemplo de simetria no seguinte quebra-cabeça também extraído da documentação do Mozart, o qual consiste em encontrar 9 dígitos (variáveis de *A* a *I*) distintos diferentes de zero tais que:

$$\frac{A}{BC} + \frac{D}{EF} + \frac{G}{HI} = 1.$$

Nesse caso, a simetria está no fato de que todas as permutações sobre qualquer conjunto de três frações (satisfatório ou não) respondem equivalentemente ao problema. Em sendo as soluções realmente equivalentes, podemos nos contentar com apenas uma em cada classe de equivalência. Uma forma freqüente de quebrar simetrias – ou seja, evitar a busca por soluções alternativas equivalentes – é introduzir, no modelo, restrições/propagadores estabelecendo relações de ordem artificiais. No exemplo acima, bastaria impor:

$$\frac{A}{BC} \geq \frac{D}{EF} \geq \frac{G}{HI}$$

Em alguns casos, outro recurso de modelagem predisponente a melhor propagação é introduzir algumas **restrições redundantes**, ou seja, teoremas das restrições originais do modelo. No exemplo acima, impor as seguintes restrições redundantes fortalece a propagação:

$$3\frac{A}{BC} \geq 1 \quad \text{e} \quad 3\frac{G}{HI} \leq 1.$$

Criação de modelo

Um sistema computacional real desenvolvido com PCR geralmente não resolve apenas um problema fixo como o exemplo das três frações acima, mas uma classe de problemas que é instanciada a partir da entrada corrente. Portanto, um passo anterior à resolução propriamente dita consiste exatamente em gerar, a partir da entrada, a descrição do problema a ser resolvido. Esse passo fundamental é conhecido como **criação de modelo**.

Um modelo estabelece uma relação lógica entre a entrada e a saída do sistema. Em conseqüência, um novo modelo é gerado a cada par de entrada e saída, envolvendo um processamento procedimental da entrada. Nessas condições, em que os detalhes do modelo a ser resolvido são gerados em tempo de execução, as tarefas de eliminação de simetrias e imposição de restrições redundantes podem se tornar complexas e até inviáveis.

Como facilmente se depreende desta breve introdução à PCR, a modelagem é uma questão central na aplicação desse paradigma. Apresentaremos, no próximo capítulo, um apanhado de noções elementares de modelagem úteis na área de PLN.

III Modelos por Restrições para a Geração de Língua Natural

Neste capítulo, apresentamos os resultados de uma revisão bibliográfica dos modelos de GLN baseados em restrições. Consideramos relevantes todas e apenas as publicações coletadas que tratassem de modelagem tal como definimos (pág. 15), ou seja, que procurassem formular **Problemas de Satisfação de Restrições** (**PSR** ou – ainda mais freqüentemente, mesmo em português – **CSP**, do inglês *Constraint Satisfaction Problem*). É importante observar que não discriminamos trabalhos em função do método de resolução empregado, de forma que nem todos os (poucos) trabalhos apresentados envolvem PCR.

Em conformidade com a motivação apresentada no Capítulo I, pode-se notar que nenhum dos trabalhos apresentados aplica PCR ou modelagem por restrições em geral à totalidade do processo de RL. No máximo, contamos com os casos do Manati e da XDG, cujo suporte à agregação e geração de expressões referenciais é apenas uma potencialidade ainda não explorada. Entretanto, tudo indica que, nesses casos, há sérias limitações em generalidade e instanciabilidade, na melhor das hipóteses.

Apresentaremos, portanto, os modelos agrupados por tarefa enfocada em cada caso: (i) geração de expressões referenciais, com um único exemplar, devido a Gardent (2002) e aplicando PCR; (ii) agregação, com apenas dois modelos, por Power (2000) e Williams (2003) e sem aplicação de PCR; (iii) lexicalização, com três modelos implementados com PCR, por Gardent & Thater (2002), Koller & Striegnitz (2002) e Pelizzoni & Nunes (2005). Estes dois último corresponde ao Manati.

III.1 Geração de expressões referenciais

O único modelo de geração de expressões referenciais por restrições que encontramos foi desenvolvido por Gardent (2002) e implementado em Oz no sistema InDiGen[6] (*Integrated Discourse Generator*). A arquitetura global desse sistema, entretanto, não é baseada em restrições.

A autora resume o problema atacado neste trabalho como “gerar descrições definidas mínimas”. Em outras palavras, dados (i) um conjunto-universo de entidades *U*, (ii) um conjunto-universo de propriedades $\wp$ e (iii) uma função $\rho : U \rightarrow 2^{\wp}$ que, para cada entidade $x \in U$, define o conjunto das propriedades válidas para *x*, o problema consiste um descrever um conjunto $S \subseteq U$ de entidades em oposição a $\overline{S} = U - S$ usando o mínimo número de elementos de $\wp$.

Primeiramente, Gardent trata as componentes conjuntivas de uma tal descrição, definindo que uma dupla $D = \left(P^{+}, P^{-}\right)$ é uma **descrição distintiva básica** (não necessariamente mínima) para um $S \subseteq U$ sse:

i. $P^{+} \subseteq \bigcap_{x \in S} \rho(x)$, isto é, P^{+} seja um conjunto de propriedades válidas para todos os elementos de *S*;

ii. $P^{-} \subseteq \wp - \bigcup_{x \in S} \rho(x)$, isto é, P^{-} seja um conjunto de propriedades que não valem para nenhum elemento de *S*; e

---

[6] http://www-old.coli.uni-saarland.de/cl/projects/indigen.html.

iii. $\forall x \in \overline{S}$, $\left|\left(P^{+} - \rho(x)\right) \cup \left(P^{-} \cap \rho(x)\right)\right| > 0$, isto é, para cada elemento *x* fora de *S*, existe pelo menos uma característica *c* que o distinga de todos os elementos em *S*, seja porque *c* vale para todos os elementos de *S* e não para *x*, seja porque *c* não vale para nenhum elemento de *S* e sim para *x*.

Perceba que uma tal descrição básica sempre fatora características (positivas ou negativas) comuns a todos os elementos do conjunto considerado. Entretanto, é possível que os elementos de um dado *S* não tenham propriedades em comum ou que estas não sejam suficientemente distintivas. Cumpre tratar a componente disjuntiva de uma descrição. Por isso, Gardent prossegue definindo que um conjunto de duplas $D = \left\{\left(S_1, D_1\right), \left(S_2, D_2\right), \ldots, \left(S_M, D_M\right)\right\}$ é uma **descrição distintiva geral** para um $S \subseteq U$ sse:

i. $1 \leq M \leq \# S$, porque, na pior das hipóteses, todos S_i serão unitários;

ii. $S = \bigcup_{1 \leq i \leq M} S_i$ e $\forall 1 \leq i, j \leq M \left(i \neq j \rightarrow S_i \cap S_j = \varnothing\right)$, isto é, $\left\{S_i\right\}$ é uma partição de S; e

iii. $\forall 1 \leq i \leq M$, D_i é uma descrição distintiva básica de S_i.

Gardent formula então o problema como gerar uma descrição distintiva geral *D* mínima, ou seja, tal que:

$$custo\left(D\right) = \sum_{T \in D} \left|\left(P^{+} - \rho(x)\right) \cup \left(P^{-} \cap \rho(x)\right)\right| \qquad \text{(III.1)}$$

seja mínimo, onde $descr\left(T\right)$ denota o segundo componente de uma tupla *T*, e $pos\left(D\right) / neg\left(D\right)$ denota o primeiro/segundo componente de uma descrição distintiva básica *D*.

O modelo é implementado em Oz de forma bastante direta. Vale notar que (III.1) é implementada como uma restrição que relaciona a variável $custo(D)$ com os conjuntos de propriedades (variáveis de interesse) num vetor de descrições distintivas básicas. Obtém-se a minimização ao efetuar uma distribuição trivial (assumindo valores crescentes, iniciando no mínimo permitido pelo primeiro ciclo de propagação) sobre $custo(D)$. Em seguida, executa-se distribuição sobre as variáveis de interesse. Enquanto os valores atribuídos a $custo(D)$ forem baixos demais, a busca falhará e tentará o próximo valor até que se chegue à primeira solução. Trata-se da materialização, em PCR, da técnica de *iterative deepening* (Russel & Norvig, 1995) da resolução por busca.

Por fim, merece atenção a codificação das propriedades, que são derivadas de um conjunto de relações semânticas situacionais sem aninhamento **(*flat*)** do tipo $f(e_1, e_2, \ldots, e_n)$, onde f é um átomo característico da relação e e_i são valores atômicos denotando as entidades relacionadas. Para cada relação $f(e_1, e_2, \ldots, e_n)$, cria-se um conjunto de n propriedades $\{f(\sim, e_2, \ldots, e_n), f(e_1, \sim, \ldots, e_n), f(e_1, e_2, \ldots, \sim)\}$, cada qual representando a relação inicial parametrizada para o relacionando substituído pelo símbolo ~. O universo $\wp$ é definido como a união de todos esses conjuntos. Como e_i não são variáveis, mas identificadores conhecidos, cada elemento de $\wp$ pode ser considerado uma constante e codificado como um numero natural exclusivo. Note que uma nova codificação deve ser estabelecida a cada novo conjunto de relações situacionais.

III.2 Agregação

Ambos os trabalhos relacionados com agregação que encontramos partem de uma estrutura retórica, segundo o modelo **RST** (Mann & Thompson, 1988), do texto a ser produzido. Uma estrutura RST é sempre uma árvore binária não-ordenada cujos nós-folha contêm o conteúdo semântico propriamente dito em algum formalismo ortogonal e os nós interiores representam

instâncias de um conjunto reduzido de relações retóricas binárias. Cada nó interior é rotulado de forma a identificar o tipo da relação que estabelece entre seus dois filhos, os relacionandos. Toda relação RST é polarizada, na medida em que sempre distribui entre seus relacionandos os papéis de **núcleo** e **satélite**, o que fica registrado na estrutura por meio de rótulos de aresta.

Tanto em (Power, 2000) quanto em (Williams, 2003), o problema tratado pode ser expresso abstratamente como determinar:

a) **ordenação**, isto é, a ordem a ser estabelecida entre nós-folha de forma a refletir a apresentação da informação no texto final. Em ambos os trabalhos, esse problema se reduz a ordenar os filhos de cada nó, ou seja, estabelecer qual vem primeiro: o núcleo ou o satélite. A ordenação final é obtida pela uma enumeração in-ordem dos nós-folha. Em termos de modelagem, tratar a ordenação corresponde a associar a cada nó interior *N* da estrutura RST de entrada uma variável de interesse $ordem(N)$ com domínio inicial $\{$núcleo-satélite, satélite-núcleo$\}$;

b) **coesão**, inserindo especificamente conjunções e advérbios conectivos ("portanto", "em conseqüência", "no entanto"). Em nenhum caso pronomes relativos são gerados como elementos de coesão, ou seja, esses modelos não determinam se uma oração subordinada adjetiva será gerada ou não. Se isso acontece, trata-se de tarefa delegada a outro módulo. Aqui o problema se reduz a determinar, para cada nó interior, um elemento conectivo (possivelmente nulo) e sua posição relativa aos trechos de texto dos nós-filho (antes ou depois de ambos ou entre eles). O **trecho de texto** de um nó *N* é o texto a ser gerado correspondente à subárvore que tem *N* como raiz. Em termos de modelagem, tratar a coesão corresponde a associar a cada nó interior *N* da estrutura RST de entrada duas variáveis de interesse $conectivo(N)$ e $posiçãoConectivo(N)$. O domínio inicial para a variável $conectivo(N)$ pode ser determinado a partir da relação retórica estabelecida por *N*;

c) pontuação. Nesse caso, o problema se reduz a determinar, para cada nó interior, qual a pontuação (possivelmente nula) a ser inserida entre os trechos de texto dos nós-filho. Em termos de modelagem, tratar a pontuação corresponde a associar a cada nó interior N da estrutura RST de entrada uma variável de interesse $pontuação(N)$, cujo domínio inicial pode ser o conjunto de todos os possíveis sinais de pontuação, incluindo um valor nulo;

Por fim, apenas Power está preocupado em determinar um nível mais refinado de:

d) estruturação, associando ainda mais duas variáveis de interesse a cada nó N (desta vez, possivelmente um nó-folha) da estrutura RST de entrada, a saber:

i. $textLevel(N)$, assumindo um dos possíveis valores ordenados:

$$section > paragraph > textSentence > textClause > textPhrase$$

cuja interpretação, *grosso modo*, é respectivamente "seção" (conjunto de *paragraphs*), "parágrafo" (conjunto de *textSentences*), "string delimitada por pontuação de fim de sentença" (conjunto de *textClauses*), "string delimitada por ponto-e-vírgula" (conjunto de *textPhrases*) e "string a ser articulada sintaticamente"; e

ii. $indentation(N)$, assumindo valores numéricos a partir de 0 e indicando se o trecho de texto de N pertence ao corpo de texto mais externo (valor 0) ou a um item (valor 1) ou subitem (2 em diante) de uma lista com possivelmente vários níveis.

Apesar das diversas semelhanças, esses trabalhos diferem radicalmente em dois quesitos, a saber:

- **origem das restrições.** Procedendo de forma mais tradicional, Power elabora manualmente uma série de restrições de boa-formação relacionando, por exemplo, (i) as diversas variáveis de cada nó-pai e com as de seus nós-filho e (ii) $conectivo(N)$ com $ordem(N)$, já que

há conectivos cujo uso restringe a ordenação entre núcleo e satélite. Por exemplo, "em conseqüência" determina que o satélite seja realizado primeiro.

Williams, por sua vez, aplica uma abordagem estatística, levantando suas restrições a partir de um córpus anotado segundo a RST. Para cada relação de rótulo *Rel* observada, tem-se uma atribuição de variáveis $f: Vars \rightarrow U$, onde $Vars = \{nucLen, satLen\} \cup V$, isto é, o conjunto *V* de variáveis de interesse já mencionadas mais as "variáveis" correlacionadas *nucLen* e *satLen*, cada qual denotando a classificação, como *curto* ou *longo*, do trecho de texto do núcleo ou do satélite, respectivamente. Numa base específica para *Rel* – $base(Rel)$ – são acumuladas as freqüências de coocorrência dos valores de cada par de variáveis em *Vars*, ou seja, a freqüência de cada $\{(x, f(x)), (y, f(y))\}$ para todo $x, y \in Vars$, $x \neq y$;

- **estratégia de resolução.** Ambos os trabalhos procuram por um ótimo. Power enumera todas as possíveis soluções para o modelo global e, em seguida, classifica-as de acordo com uma medida de qualidade, selecionando a ótima. Nisso, reporta que o número de soluções *válidas* se aproxima usualmente de 5^n, onde *n* é o número de nós-folha da estrutura de entrada.

  Williams, por sua vez, realiza um busca gulosa e tira proveito do fato de que os nós-folha de sua estrutura RST de entrada já são trechos finais de texto produzidos por um estágio anterior de sua arquitetura em *pipeline*. Iterando pela estrutura em ordem decrescente de profundidade, seu sistema GIRL resolve, para cada nó interior *N* isoladamente, um CSP sobre as variáveis $Vars = \{nucLen, satLen\} \cup V$, onde *nucLen* e *satLen* estão determinadas com os valores observados para *N*, e uma atribuição *f* é solução se e somente se, para todo par de variáveis distintas $x, y \in Vars$, a coocorrência $\{(x, f(x)), (y, f(y))\}$ é sancionada

por $base(rel(N))$. GIRL obtém todas as soluções para esse CSP e assume a solução mais freqüente.

III.3 Lexicalização

A lexicalização implica, no mínimo, (i) escolha lexical e (ii) montagem de uma estrutura sintática. Se, por um lado, já chegamos à conclusão de que a escolha lexical tem expressão natural dentro da PCR por meio de restrições de seleção; por outro lado, não podemos sequer dizer que o modelo de análise sintática do Capítulo III monta uma estrutura. É exatamente a modelagem da sintaxe ou, mais abstratamente, a modelagem de estruturas, questão fundamental da lexicalização por PCR, que enfatizaremos nesta seção.

Em primeiro lugar, é importante adotar uma representação que permita que a montagem da estrutura se dê de forma monotônica. Monotonicidade é o pré-requisito por excelência da propagação e do paradigma como um todo. Em específico, é preciso abandonar qualquer abordagem transformacional em que, por exemplo, uma aresta entre dois nós tenha de ser apagada. Portanto, formalismos muito populares em GLN como **Gramáticas de Adjunção de Árvores** (**TAG**, do inglês *Tree Adjoining Grammars*) não rendem bons modelos. Pelo menos, não diretamente.

Revisaremos a seguir os quatro modelos de lexicalização por PCR presentes na literatura, a saber: (Gardent & Thater, 2001), recorrendo a Gramáticas de Descrições Arbóreas; (Koller & Striegnitz, 2002), equacionando TAGs (!) por meio de Gramáticas de Dependências; e (Pelizzoni & Nunes, 2005) e (Debusmann et al., 2004a), recorrendo diretamente a Gramáticas de Dependências. Esses quatro trabalhos estão profundamente relacionados e são apresentados numa seqüência que pode ser vista como uma evolução em direção a maior propagação.

Gardent &Thater (2001) – geração com uma gramática baseada em descrições arbóreas

Se as TAGs não servem para modelagem por restrições porque seus blocos de montagem são árvores a serem *transformadas* e *combinadas* (operações não-monotônicas), quando tomamos por blocos *descrições subespecificadas dessas mesmas árvores*, tudo passa a funcionar perfeitamente. Tal é a abordagem de Gardent & Thater (2001), que adaptaram para geração o que Duchier & Thater (1999) tinham feito para análise sintática.

As gramáticas usadas por Gardent & Thater (2001) são conhecidas por **Gramáticas de Descrições Arbóreas** (**TDGs**, do inglês *Tree Description Grammar*s). A idéia de valência, já embrionária em nosso exemplo do Capítulo III, é agora levada a sério, tanto que as árvores montadas são ditas **eletrostáticas**. Trata-se de um formalismo gramatical lexicalizado, cujo léxico associa cada símbolo terminal (no caso da geração, um literal lógico *flat*; no caso da análise, uma palavra) com uma lista de Descrições de Árvore Eletrostática (DAEs) alternativas. Cada DAE é uma fórmula lógica que descreve *parcialmente* uma (sub)árvore.

Apresentaremos a definição formal de DAE reproduzindo, em parte, o (excelente) trabalho de Duchier & Thater (1999). Assumindo três conjuntos infinitos e disjuntos $Vars^0$, $Vars^+$ e $Vars^-$ de variáveis (respectivamente, com carga nula, positiva e negativa), e um conjunto L de rótulos (que pode, por exemplo, ser um tipo de AVM, de modo a abstrair a componente morfossintática do modelo) uma **Descrição de Árvore Eletrostática (DAE)** é uma fórmula da seguinte forma:

$$\phi \ ::= \ x\,R\,y \mid x:\langle y_1,\ldots,y_n\rangle \mid x \leftarrow l \mid \phi_1 \wedge \phi_2 .$$

Nessa gramática, x, y e y_i denotam variáveis a serem interpretadas como os nós de uma árvore-solução; e R, possíveis relações arbóreas que devem valer entre duas variáveis. R pode ser

construído como uma combinação booleana de $=, \lhd^+, \rhd^+, \prec, \succ$ – onde $\lhd^+$ denota dominância estrita[7]; e $\prec$, precedência – segundo a seguinte gramática:

$$R \quad ::= \quad = \mid \lhd^+ \mid \rhd^+ \mid \prec \mid \succ \mid R_1 \cup R_2 \mid R_1 \cap R_2 \mid \neg R\,.$$

A expressão $x : \langle y_1, \ldots, y_n \rangle$ estabelece que $\{y_i\}$ são os filhos imediatos de *x*; e, por fim, $x \leftarrow l$ denota que *x* é rotulado com *l*.

Dada uma nova fórmula semântica *Sem* a ser verbalizada, parte da construção do modelo corresponde a construir, a partir das DAEs do léxico, um sistema de equações cujas soluções *conterão* (vide abaixo) as árvores sintáticas que verbalizam *Sem*. Para cada literal de *Sem*, o léxico fornece uma disjunção de DAEs; e o sistema surge como a conjunção dessas disjunções. Na construção do modelo, é importante observar que:

a) inicia-se com uma DAE axiomática $x_0 \leftarrow S$, onde $x_0 \in Vars^-$, e S é um rótulo especial correspondente ao símbolo inicial da gramática;

b) toda DAE resultante da consulta ao léxico contém variáveis exclusivas, identificadas com números naturais também exclusivos dentro do sistema. Daí os conjuntos de variáveis serem teoricamente infinitos;

c) essas variáveis, portanto, não serão as “variáveis de interesse” para a PCR, sendo antes manipuladas como constantes numéricas quando da resolução propriamente dita. Para evitar confusão entre as variáveis das DAE e as reais variáveis (de interesse ou não) do modelo que está sendo criado, referir-nos-emos àquelas como **VDAEs**, deste ponto em diante;

---

[7] A expressão $x \lhd^+ y$ estabelece que *y* deve estar na subárvore cuja raiz é *x*, mas não como raiz.

d) em conseqüência, podemos definir uma matriz *Vars* totalmente determinada tal que $Vars_{i,j}$ denote o conjunto de VDAEs da *j*-ésima DAE alternativa do *i*-ésimo literal da entrada. Vale notar que a matriz *Vars* cobre também a DAE axiomática em (a), numa linha de ordem 0;

e) a disjunção das DAEs alternativas para o *i*-ésimo literal da entrada pode ser implementada como uma simples restrição de seleção dentro das linhas de *Vars*, assim: $[Vars_i]\langle Sel_i \rangle = SatVars_i$;

f) apenas as VDAEs das DAEs selecionadas devem ser **saturadas**, ou seja, caso tenham carga não-nula, devem ser unificadas com uma única VDAE de carga oposta e, caso sejam neutras, devem permanecer não-unificadas. O conjunto das VDAEs a serem **saturadas** é dado por $SatVars = SatVars_0 \cup \ldots \cup SatVars_n$, onde *n* é o número de literais da entrada;

g) pode-se reificar por uma matriz $B_{i,j}$ a seleção da *j*-ésima DAE alternativa do *i*-ésimo literal da entrada, assim:

$$dom(B_{i,j}) = \{0,1\} \wedge \left(B_{i,j} = 1 \leftrightarrow Sel_i = j\right);$$

h) para toda VDAE *x* pertencente a $Vars_{i,j}$ definimos, por conveniência:

$$sel(x) = B_{i,j};$$

i) para poder identificar a carga das variáveis são definidos quatro conjuntos constantes, a saber: $V = \bigcup_{i,j} Vars_{i,j}$, $V^+ = V \cap Vars^+$, $V^- = V \cap Vars^-$ e $V^0 = V \cap Vars^0$;

j) nossas variáveis de interesse são, na verdade, conjuntos $eq(x)$ definidos para cada VDAE *x*. Cada $eq(x)$ inclui *x* e quaisquer VDAE que unificam com *x* na solução, ou seja, cuja

interpretação (nó correspondente numa árvore-solução) coincide com a de *x*. As restrições sobre uma tal variável de interesse são as seguintes:

i. $eq(x) = \{x\} \uplus mates(x)$, onde $\uplus$ é a união disjunta e $mates(x)$ é definido a seguir por meio de restrições reificadas;

ii. $x \in V^0 \rightarrow |mates(x)| = 0$, ou seja, VDAEs neutras não unificam;

iii. $x \notin V^0 \rightarrow |mates(x)| = sel(x)$, ou seja, se x tem carga não-nula, seu conjunto de unificantes $mates(x)$ é unitário se e somente se x participa de alguma DAE selecionada; caso contrário, $mates(x)$ é vazio;

iv. $x \in V^+ \rightarrow mates(x) \subseteq V^-$ e, reciprocamente, $x \in V^- \rightarrow mates(x) \subseteq V^+$, ou seja, VDAEs com carga não-nula só unificam com VDAEs de carga oposta;

k) por fim, para maximizar a propagação, *todas* as restrições correspondentes às DAE – selecionadas ou não – são impostas e procura-se por uma atribuição satisfatória para *todas* as $eq(x)$. Determinados os valores, temos a raiz da árvore sintática gerada em $eq(x_0) - \{x_0\}$, onde x_0 é a VDAE do axioma em (a).

Para uma formalização completa dessa modelagem, consulte (Duchier & Thater, 1999) e (Gardent & Thater, 2001).

Melhorando a propagação: Gramáticas de Dependências

Engenhosa e elegante que seja, a modelagem por TDGs não rende boa propagação, de forma que, para gramáticas e entradas maiores, a explosão combinatória logo se faz sentir. Para resolver isso, uma nova solução em modelagem foi adotada e corresponde ao estado da arte hoje, a saber: **Gramáticas de Dependência** (**DGs**, do inglês *Dependency Grammars*). São duas as idéias básicas da passagem de TDGs para DGs, a saber:

a) eliminação de nós não-terminais: análises agora são dígrafos conexos com arestas rotuladas e possivelmente restritos para **arboreidade** (qualidade de árvore, tradução do inglês *treeness*). Entretanto, inexiste a idéia de nó não-terminal. Numa análise sintática, por exemplo, cada nó representa diretamente uma palavra da entrada e as arestas representam relações sintáticas entre as palavras. Naturalmente, uma tal árvore sintática é, em geral, não-binária;

b) pulverização de carga ou valencial: em vez de cargas positiva, negativa e nula apenas, as DGs permitem definir um conjunto arbitrário de traços valenciais *Val* e estabelecer, para cada nó *x* da estrutura a ser montada, duas valências $in(x)$ – dita de entrada – e $out(x)$ – dita de saída – cujos elementos pertencem a $Q \times Val$, onde $Q = \{1, ?, +, *\}$ é o conjunto de **quantificadores**. Para que uma análise *G* seja bem-formada, é necessário que:

i. para toda aresta $(x, y, v) \in G$, $Op \times \{v\} \cap out(x) \neq \varnothing$ e $Op \times \{v\} \cap in(y) \neq \varnothing$, ou seja, toda aresta de *x* para *y* deve ser sancionada pelas valências de saída de *x* e de entrada de *y*;

ii. para todo *x* e toda dupla $(_, v) \in (\{1, +\} \times Val) \cap in(x)$, $\left|\{y : (y, x, v) \in G\}\right| > 0$, ou seja, os quantificadores 1 e + exigem que a valência de entrada seja satisfeita pelo menos uma vez; analogamente

iii. para todo *x* e toda dupla $(_, v) \in (\{1, +\} \times Val) \cap out(x)$, $\left|\{y : (x, y, v) \in G\}\right| > 0$;

iv. para todo *x* e toda dupla $(_, v) \in (\{1, ?\} \times Val) \cap in(x)$, $\left|\{y : (y, x, v) \in G\}\right| \leq 1$, ou seja, os quantificadores 1 e ? exigem que a valência de entrada seja satisfeita no máximo uma vez; analogamente

v. para todo *x* e toda dupla $(_, v) \in (\{1, ?\} \times Val) \cap out(x)$, $\left|\{y : (x, y, v) \in G\}\right| \leq 1$.

Essas características conferem maior propagação e permitem que o léxico não mais retorne DAEs alternativas numa consulta, mas tuplas (l, In, Out), onde *l* é um rótulo de nó; e *In* e *Out*, valências de entrada e saída, respectivamente.

Esse tipo de formalismo gramatical e modelagem foi introduzido em versão preliminar por Duchier (1999), para a análise sintática, e subseqüentemente amadurecido por Duchier & Debusmann (2001) e Duchier (2002), em ambos os casos para dar conta da ordem de constituintes, inclusive com vantagens sobre os formalismos convencionais para línguas com ordenação mais livre, como o alemão.

Koller & Striegnitz (2002) – geração como análise sintática

Como uma primeira tentativa de usar a nova abordagem de DGs para a geração, temos o trabalho de Koller & Striegnitz (2002), que operam com um tipo especial de TAGs (valencial) por meio de DGs da seguinte forma:

a) dada uma conjunção de literais semânticos *flat* a ser verbalizada, as árvores básicas alternativas para cada literal são recuperadas de acordo com o léxico da TAG vigente;

b) como essas árvores embutem a idéia de valência, há um procedimento para, neste momento, (i) montar um conjunto de traços valenciais *Val* e, para cada árvore alternativa de cada literal *x*, (ii) especificar valências $in(x)$ e $out(x)$ de uma respectiva acepção de *x* numa DG hipotética. Para todos os efeitos, essa DG está sendo construída por esse procedimento;

c) em seguida, a seqüência de literais é *analisada sintaticamente* como descrito em Duchier (2002) de acordo com a DG recém-construída;

d) a partir da árvore sintática T_{DG} assim obtida, há um procedimento para reconstruir a árvore correspondente T_{TAG} segundo o modelo TAG. Esse procedimento é possível porque, em

T_{DG}, apenas uma acepção está ativa para cada literal; e, como as acepções estão associadas biunivocamente com uma árvore básica da TAG original, há um mapeamento *tree* entre cada literal x e uma dessas árvores. Além disso, o conjunto de valências *Val* é construído de tal forma que toda aresta (x, y, v) em T_{DG} especifica, pelo rótulo v, uma operação de combinação a ser realizada entre $tree(x)$ e $tree(y)$. Com efeito, T_{DG} consiste numa *árvore de derivação sintática* segundo o modelo TAG.

Apesar de render boa propagação, ser admirável pela engenhosidade e ter seu valor histórico, essa abordagem insere apenas uma complicação a mais. Já que, em última análise, toda a TAG utilizada é reduzida a uma DG, parece mais razoável usar este último formalismo diretamente. Além disso, o último estágio evolutivo do formalismo de DG – a XDG, apresentada a seguir – oferece possibilidades não cobertas pela abordagem de Koller & Striegnitz.

Pelizzoni & Nunes (2005) – Deconversão UNL por meio de programação multiparadigmática

Em (Pelizzoni & Nunes, 2005), apresentamos e motivamos o **Manati**, nosso modelo de deconversão UNL. A **UNL** – do inglês *Universal Networking Language* (Martins, 2004; Martins et al., 2002; Martins et al., 2000) – é um formalismo de representação semântica originalmente projetado para servir de interlíngua em sistemas de tradução automática. A tarefa de **deconversão UNL** para uma dada língua-alvo L consiste exatamente em transformar expressões UNL no texto correspondente em L.

UNL. A UNL é uma representação usualmente monossentencial e tenta capturar o significado de uma dada sentença por meio de um hiperdígrafo, onde (i) nós básicos se referem a instâncias de “conceitos universais” ou Palavras Universais (**UW**, do inglês *Universal Words*), (ii) arestas rotuladas estabelecem relações binárias entre os referentes dos nós, (iii) hipernós per-

mitem a construção de “conceitos complexos”, e (iv) atributos[8] de nós permitem, por exemplo, expressar modalização e determinação de conceitos. Embora uma explicação aprofundada da UNL fuja ao escopo desta monografia, vale notar que:

- as UWs têm classe aberta, enquanto relações e atributos têm classe fechada, provindo de um conjunto restrito de rótulos;
- todo hipernó – incluindo o mais externo (e implícito) abrangendo o todo de um hiperdígrafo UNL – tem exatamente um nó de entrada, marcado explicitamente com o atributo ***@entry***, a partir do qual a deconversão se inicia. Nós de entrada correspondem aproximadamente a núcleos clausulares em línguas naturais.

Implementação. O Manati também é o nome da implementação de nosso modelo, feita completamente em Oz. Entretanto, o Manati não é uma aplicação, mas um *framework* de software, ou melhor, simplesmente uma biblioteca para servir de base para módulos/sistemas de deconversão UNL. Isso não deve ser considerado uma desvantagem, já que é particularmente favorável à interoperabilidade do Manati com outros módulos, o que amplia seu espectro de aplicação.

---

[8] Os atributos de um nó UNL não têm valores como os atributos de um AVM, constituindo, na realidade, apenas um conjunto de valores associado ao nó em questão.

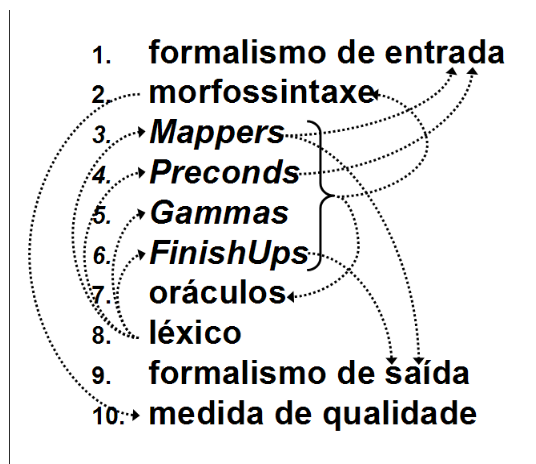

Figura 2: dependências entre os parâmetros do Manati. Uma seta de *X* para *Y* indica que *X* é definido em termos de *Y*

Parâmetros. Configurar o Manati significa preencher dez parâmetros, a saber:

1. **formalismo de entrada**, que não precisa ser necessariamente a UNL, mas deve ser um tipo de hiperdígrafos, com conjuntos de relações e atributos definidos pelo instanciador e sempre respeitando o axioma de entrada (atributo *@entry*);

2. **morfossintaxe,** como uma hierarquia de tipos de AVM definindo classes gramaticais (**POS**, do inglês *Part of Speech*) e seus respectivos traços morfossintáticos. A definição da morfossintaxe é apoiada por uma linguagem conveniente que recorre aos mecanismos de codificação de domínios apresentados no Capítulo III;

3. **mapeamento sintático**, pela definição de uma hierarquia de ***Mappers***, classes (na acepção usual segundo o paradigma de objetos) de mapeadores sintáticos. Um *Mapper* procura um padrão no (hiperdí)grafo de entrada e constrói parte da saída como uma árvore sintática (segundo o modelo DG) sem considerar restrições morfossintáticas, como concordância, ou de ordem entre constituintes. *Grosso modo*, a função de um *Mapper* se resume a mapear papéis semânticos em sintáticos. Como veremos, é extremamente interessante maximizar a *fatoração* de *Mappers*, ou seja, que os *Mappers* presentes nas regras de tradução para uma mesma UW coincidam;

4. **precondições semânticas**, pela definição de uma hieraquia de ***Preconds***, classes de precondições semânticas a serem verificadas antes da aplicação dos *Mappers*, para favorecer a fatoração destes;

5. **restrições morfossintáticas**, pela definição de uma hierarquia de *Gammas*, classes que impõem restrições entre as matrizes de traços morfossintáticos de cada par de nós $(Pai, Filho)$ da árvore em construção;

6. **restrições de ordem**, pela definição de uma hierarquia de *FinishUps*, classes cujos métodos são ativados quando uma subárvore acaba de ser construída (mais detalhes abaixo);

7. um número arbitrário de **oráculos** – ou seja, módulos externos tais como bases de conhecimento e interfaces com o usuário – a serem consultados a praticamente qualquer momento da criação de modelo;

8. **léxico**, na verdade, uma base de regras de tradução, cada uma de cujas entradas é uma tupla $R = (UW, TranList, NodeType, POS, Precond, Mapper, Gamma, FinishUp)$, onde *UW* é a UW que *R* traduz; *TransList*, uma lista de possíveis traduções para *UW*; *NodeType*, a classe do nó sintático a ser criado implicitamente como raiz da árvore construída pela aplicação de *R*; e os demais elementos são classes pertencentes às hierarquias implicadas por seus nomes. Os elementos dessas quatro hierarquias ortogonais são livremente combinados na construção das regras de tradução;

9. **formalismo de saída**, que pode incluir anotações diversas, como a parentização sintática;

10. uma **medida de qualidade** a ser maximizada para dirigir a busca por uma solução.

A Figura 2 apresenta as dependências entre os parâmetros do Manati. Nessa figura, uma seta de um parâmetro *X* para outro *Y* indica que *X* depende de *Y*, ou seja, que a definição de *X* é feita em termos do que está definido para *Y*.

Qualidade. A medida de qualidade é fornecida ao modelo na forma de um predicado binário *Q* impositor de uma restrição de superioridade qualitativa. Durante a busca, *Q* é iterativamen-

te aplicado a pares $(CurBest, Wannabe)$, onde *CurBest* é a melhor solução encontrada até o momento, e *Wannabe* é um modelo parcialmente determinado que vai tentar ser ainda melhor que *CurBest*. Na verdade, é exatamente a imposição de $Q(CurBest, Wannabe)$ que dá a *Wannabe* o ímpeto de ser melhor, ao restringir o cômputo de qualidade de *Wannabe* para que seja estritamente maior que o de *CurBest*. Como *CurBest* e *Wannabe* são dados como as raízes de duas árvores sintáticas, e *Wannabe* não está completamente determinado, o que impede o acesso direto a suas subárvores, a restrição imposta por *Q* deve relacionar ambos os modelos tão-somente em termos das variáveis presentes nas duas raízes. Portanto, para medidas de qualidade complexas, espera-se que o instanciador inclua traços relativos a qualidade nas POSs e promova a sua propagação em direção à raiz por meio de restrições especiais impostas por *Gammas* e *FinishUps*.

Até o momento, experimentamos apenas com a minimização do comprimento da saída, o que é especialmente interessante para a geração da Língua Brasileira de Sinais (LIBRAS) porque, nesta língua, é bastante freqüente que duas ou mais palavras distintas possam ser combinadas em uma única forma preferencial (Brito, 1995). Como o Manati fornece, para cada nó sintático, um atributo *yieldCard* denotando o número palavras que sua subárvore carrega, a medida de qualidade "menor é melhor" é otimizada pelo seguinte predicado simples:

```
proc {LessIsMore CurBest Wannabe}
    Wannabe.yieldCard <: CurBest.yieldCard
end
```

Alternativamente, se alguém está interessado numa única solução qualquer, é possível selecionar a primeira obtida por meio do seguinte predicado, ainda mais simples:

```
proc {FirstWillDo _ _}
    fail
end
```

Criação de modelo. A criação de modelo no Manati começa com uma requisição de deconversão implícita para o nó *@entry* de um grafo UNL de entrada. Quando a deconversão de um *nó-fonte* (i.e. semântico) *Src* é requisitada, sua UW é usada para recuperar do léxico regras de tradução aplicáveis em princípio. Para cada regra *R*, *Precond* é ativado, realiza a verificação

semântica especificada e, sse *TransList* tem mais de um elemento, deve selecionar exatamente um deles para ser a palavra-alvo da regra. A *palavra nula* também pode ser elemento de *TransList*, criando um nó sintático invisível e possibilitando categorias nulas.

Deste ponto em diante, o Manati tenta otimizar a aplicação de PCR instanciando um único *mapper* (em minúsculas, porque instância) para cada conjunto de regras bem-sucedidas até agora que compartilhem da mesma classe *Mapper*. Cada *mapper* recebe um *nó-alvo* (i.e. sintático) *default* construído a partir dos demais dados do conjunto de regras que lhe deram origem, i.e. palavras-alvo, *NodeTypes*, *POSs*, *Gammas* e *FinishUps*. Trata-se, na verdade, de um complexo nó seletor de dois níveis criado por meio de restrições de seleção. Cabe a cada *mapper* decidir o que fazer com seu respectivo nó-alvo *default*: (i) simplesmente ignorá-lo (o que não representa desperdício devido a *lazy evaluation*), (ii) retorná-lo como raiz após construir-lhe uma subárvore ou, similarmente, (iii) usá-lo como parte de qualquer estrutura sintática que venha a construir.

Cada *mapper* (i) tenta reconhecer um subgrafo específico a partir do nó-fonte *Src*, (ii) cria um conjunto de nós-alvo (geralmente correspondendo a palavras na língua-alvo), (iii) estabelece relações sintáticas binárias entre eles, momento em que as restrições *Gamma* são impostas, e, entre outras operações avançadas, (iv) possivelmente *edita* o grafo de entrada *de forma estritamente local*, ou seja, visível apenas para os *mappers* que lhe são subordinados.

Alguns nós-alvo em (ii) podem ser criados por delegação do *mapper* corrente, ou seja, aplicando-se recursivamente o mesmo processo a alguns nós-fonte do subgrafo reconhecido em (i). Isso é possível porque cada requisição de deconversão (recursiva ou não) retorna exatamente um único nó-alvo, que bem pode ser um nó seletor escolhendo entre as raízes de várias subárvores mutuamente excludentes produzidas por *mappers* alternativos.

Quando um *mapper* termina, restrições *FinishUp* são aplicadas à raiz *Target* a ser retornada. Esse é um momento importante porque todos os nós-filho de *Target* são conhecidos e fica facilitada a imposição de certas restrições de ordenação.

Busca. Após a criação do modelo, o Manati começa a buscar por uma solução ótima. Esse processo é dirigido por três atributos importantes dos nós-alvo, quais sejam: dado um nó alvo *T*, *T.id* é a posição absoluta de *T* no texto gerado; *T.active*, um booleano indicando se a subárvore de *T* realmente participa da solução corrente ou é descartada; e, para *nós seletores de primeira ordem* apenas, gerados para um cojunto de regras coincidentes para $(UW, Mapper, Gamma, FinishUp)$ e que selecionam de uma lista de palavras-alvo alternativas, *T.lexI* denota o índice da palavra escolhida.

O *script* de busca é o seguinte (lembre-se de que cada passo de distribuição espera que a propagação chegue à estabilidade):

1. distribuir sobre o vetor de todos os atributos *active*, priorizando (i) ativação sobre desativação e (ii) elementos que aparecem antes no vetor, o que corresponde aproximadamente à ordem em que as regras de tradução aparecem no léxico;

2. *ActiveNds* ← vetor de todos os nós seletores de primeira ordem;

3. para todo *Id* em $[ActiveNds.id]$, impor $dom(Id) \subseteq arity(ActiveNds)$;

4. distribuir ingenuamente sobre $[ActiveNds.lexI]$, i.e., selecionando a primeira variável não-determinada e experimentando valores mínimos primeiro;

5. distribuir sobre $[ActiveNds.id]$ usando a estratégia *first-fail*;

6. se uma solução *CurBest* for encontrada, tentar melhorá-la (i) tomando um novo modelo *Wannabe*, (ii) impondo $Q(CurBest, Wannabe)$ para um predicado Q definindo superioridade qualitativa e (iii) reiterando a busca.

Complexidade. A maior falha do Manati reside em sua dependência em relação à fatoração de *Mappers*: quanto menos fatoração for realizada, tanto mais a complexidade em *tempo e tamanho* da criação de modelo se aproximará da exponencial, o que inviabiliza completamente sua aplicação. Para evitar isso, o Manati embute vários mecanismos para causar a falha precoce de *mappers* e assim evitar sua proliferação. Entretanto, a eficácia tanto da fatoração quanto desses mecanismos numa aplicação real ainda não foi atestada.

Limitações. O Manati dá muita liberdade para a definição de seus parâmetros, e é difícil dizer, no momento, qual seria seu suporte ao tratamento da agregação e geração de expressões referenciais. Entretanto, cremos que a instanciabilidade cairá bruscamente mesmo para os casos mais simples. Por outro lado, para a aplicação específica de deconversão UNL em sistemas de tradução automática, talvez o Manati seja suficiente em muitos casos, já que freqüentemente (i) a UNL é empregada sob a hipótese monossentencial, (ii) os grafos UNL gerados refletem as soluções de agregação e geração de expressões referenciais adotadas nos textos-fonte, e (iii) assume-se (bastante temerariamente) que as mesmas soluções valham também para a língua-alvo.

IV Introdução à XDG

Como as contribuições mais substanciais deste doutorado estão baseadas no formalismo/metodologia XDG e foram implementadas a partir do sistema associado XDK, importa fornecer ao leitor elementos desses dois itens que compõem o que podemos chamar nossa plataforma de trabalho. Além disso, é vital que, nesse contato, o leitor encontre evidências suficientes do amplo suporte à instanciabilidade que herdamos ao adotar essa plataforma. Tal é o propósito do presente capítulo, que deve ser considerado com parte da revisão bibliográfica e, portanto, não tratará de nenhum resultado ou contribuição deste doutorado. Pretendemos que o conteúdo ora apresentado sirva não só como subsídio para a compreensão dos capítulos seguintes, mas também como *baseline* para a apreciação justa da extensão de nossas contribuições.

Manteremos um tom geral prático e didático; para um tratamento mais detalhado e teórico, referimos o leitor a outras fontes. A tese de doutorado de Debusmann (2006) é, sem dúvida, a fonte mais amigável dos pontos de vista teórico, comparativo e ilustrativo, muito embora a formalização da XDG mais elegante e apropriada só venha a ser encontrada em artigo posterior (Debusmann, 2007). Os resultados teóricos e práticos mais interessantes e atuais estão resumidos em (Debusmann & Kuhlmann, 2007), que inclusive desmente certas impressões iniciais pessimistas quanto à eficiência do XDK emitidas em (Debusmann, 2006). Por fim, o *website* da XDG[9] reúne praticamente todas as publicações de trabalhos científicos baseados no formalismo, cobrindo diversas aplicações.

---

[9] http://www.ps.uni-sb.de/~rade/xdg.html

IV.1 Análises XDG

Como preparação para introduzir a noção de gramática nos moldes da XDG, é conveniente tratar antes da noção de análise neste modelo, que escapa consideravelmente do convencional.

Grafos

Partamos do conceito razoavelmente tradicional de **dígrafos com arcos rotulados** — aos quais, por conveniência, faremos referência daqui por diante como **grafos**, simplesmente. Formalmente, dado um conjunto $\mathbb{V}$ de vértices e um conjunto L de rótulos de aresta, nossos grafos de interesse devem ter conjuntos de arestas $E \subseteq \mathbb{V} \times \mathbb{V} \times L$, ou seja, suas arestas devem ser tuplas (v_1, v_2, l). Vale notar que essa formulação permite que dois vértices de um mesmo grafo possam estar ligados por mais de uma aresta, contanto que as arestas em questão tenham rótulos distintos.

Detenhamo-nos agora sobre os rótulos de aresta dos nossos grafos. Cada um deles é, na verdade, uma dupla de átomos $l = (r, d)$, onde *r* designa o tipo da relação estabelecida entre os vértices (correspondendo à noção usual de rótulo); e *d*, a **dimensão** em que a relação se estabelece. Por conveniência, adotaremos o seguinte apoio notacional para toda aresta $e = \big(u, v, (r, d)\big)$:

$$
\begin{aligned}
from(e) &= u \\
to(e) &= v \\
rel(e) &= r \\
dim(e) &= d
\end{aligned}
$$

onde $from(e)$, $to(e)$, $rel(e)$ e $dim(e)$ são ditos respectivamente a **origem,** o **destino**, a **relação** e a **dimensão** da aresta *e*. Definimos também os seguintes operadores relacionais de dominância direta:

$$u \overset{l}{\rightarrow}_d v \Leftrightarrow \big(u, v, (l, d)\big) \in E$$

$$u \rightarrow_d v \Leftrightarrow \exists l \left(u \overset{l}{\rightarrow}_d v \right)$$

Além disso, sempre que houver um caminho de *u* a *v* na dimensão *d* iniciando com uma aresta de rótulo *i* e terminando com uma aresta de rótulo *f*, poderemos escrever o seguinte:

$$u \lhd_d v$$

$$u \overset{i}{\rightarrow} \unlhd_d v$$

$$u \unlhd \overset{f}{\rightarrow}_d v$$

Por fim, definimos o operador de dominância relaxada da seguinte forma:

$$u \unlhd_d v \Leftrightarrow u \lhd_d v \vee u = v$$

Assumimos também a composicionalidade desses operadores relacionais, bem como a existência do operador de opcionalidade [ ], de forma que caminhos podem ser total ou parcialmente especificados, mencionando-se ou não vértices intermediários, como exemplificado na Tabela 1.

Tabela 1: exemplos de expressões compostas de caminho em grafos e seus respectivos significados

Expressão	Significado
$u \overset{i}{\rightarrow} \unlhd \overset{f}{\rightarrow}_d v$	caminho de tamanho pelo menos 2, iniciando com o rótulo *i* e terminando com *f*
$u \overset{i}{\rightarrow} \rightarrow_d v$	caminho de tamanho 2, iniciando com o rótulo *i*
$u \overset{i}{\rightarrow} [\rightarrow]_d v$	caminho de tamanho 1 ou 2, iniciando com o rótulo *i*
$u \overset{i}{\rightarrow} w \unlhd \overset{f}{\rightarrow}_d v$	aresta de *u* para *w* de rótulo *i* e caminho de tamanho pelo menos 1 de *w* para *v* terminando com o rótulo *f*

É bastante usual que grafos tenham suas arestas rotuladas para denotar relações entre objetos (cf. redes semânticas e as estruturas de dependência discutidas no Capítulo III); a novidade

aqui consiste exatamente em adicionar uma dimensão a cada relação/aresta. Isso permite agrupar facilmente arestas de rótulos distintos, mas que compartilhem de uma mesma dimensão. Dado um grafo $g = (\mathbb{V}, E)$ e uma dimensão *d*, podemos inclusive definir uma operação $\prod_d g$ — dita a **projeção** de *g* em *d* — assim:

$$\prod_d g = \left(\mathbb{V}, \{e \in E : dim(e) = d\}\right).$$

Cada $\prod_d g$ é um grafo contendo exatamente os mesmos vértices de *g*, mas todas e somente as arestas "pertencentes" à dimensão *d*. Coincidentemente, a forma mais intuitiva de visualizar um grafo é exatamente considerá-lo como a "soma" de suas projeções em todas as suas dimensões, como ilustra a Figura 3.

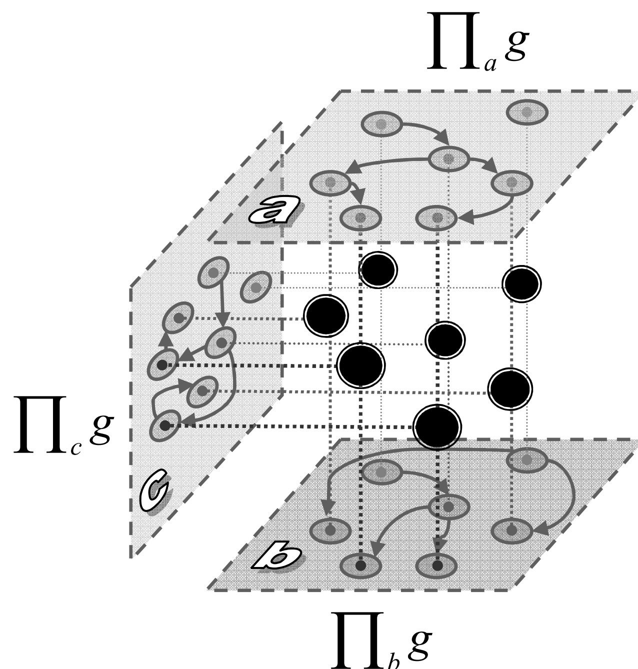

Figura 3: projeções de um grafo *g* (omitidas as relações das arestas). Se *a*, *b* e *c* constituem todas as dimensões de *g*, trata-se da visão do grafo como a soma de suas projeções

A motivação por trás desse artifício é permitir que um único grafo reúna descrições — ou análises — distintas, paralelas e complementares de um mesmo objeto. De forte apelo intuiti-

vo, tal é o cerne do que chamamos **metáfora multidimensional** da XDG. Cada uma das análises unidimensionais consiste na projeção do objeto sobre uma respectiva dimensão de descrição, ou melhor, numa visão do objeto de um ponto de vista específico. O objeto, entretanto, só fica perfeita ou suficientemente descrito quando se consideram todas as análises unidimensionais ao mesmo tempo, ou melhor, quando considerado em todas as suas dimensões.

A Figura 4 apresenta o grafo de uma análise XDG para a frase "*Peter promises Mary to smile.*" segundo a gramática `diss.ul`, desenvolvida por Debusmann (2006) para um fragmento do inglês e que integra a distribuição do XDK 1.7. Trata-se de uma saída gráfica gerada diretamente por esse sistema cuja forma de representação é exatamente a justaposição — ou "soma" — das projeções do grafo em todas as dimensões especificadas pela gramática em questão. Os identificadores das dimensões, estipulados arbitrariamente por `diss.ul`, figuram à esquerda das respectivas projeções e são os seguintes: ID (do inglês *Imediate Dominance*), IS (*Information Structure*), LP (*Linear Precedence*), PA (*Predicate Argument structure*), PS (*Prosodic Structure*) e SC (*SCope structure*).

Antes de prosseguir discutindo esse grafo, cabe uma palavra de advertência. Neste instante e ao longo deste capítulo, vamos nos ater a `diss.ul` e destacar algumas de suas características. No entanto, vale advertir desde já que essa gramática constitui um modelo que foi seguido apenas parcialmente pelo XDGen. Utilizamo-la por três motivos principais, a saber: (i) sua relativa simplicidade, (ii) sua suficiência para ilustrar a abordagem XDG e quão pouco ortodoxa ela tende a ser e (iii) sua representatividade da prática corrente em modelagem XDG, o que permitirá avaliar a novidade de nossa própria modelagem.

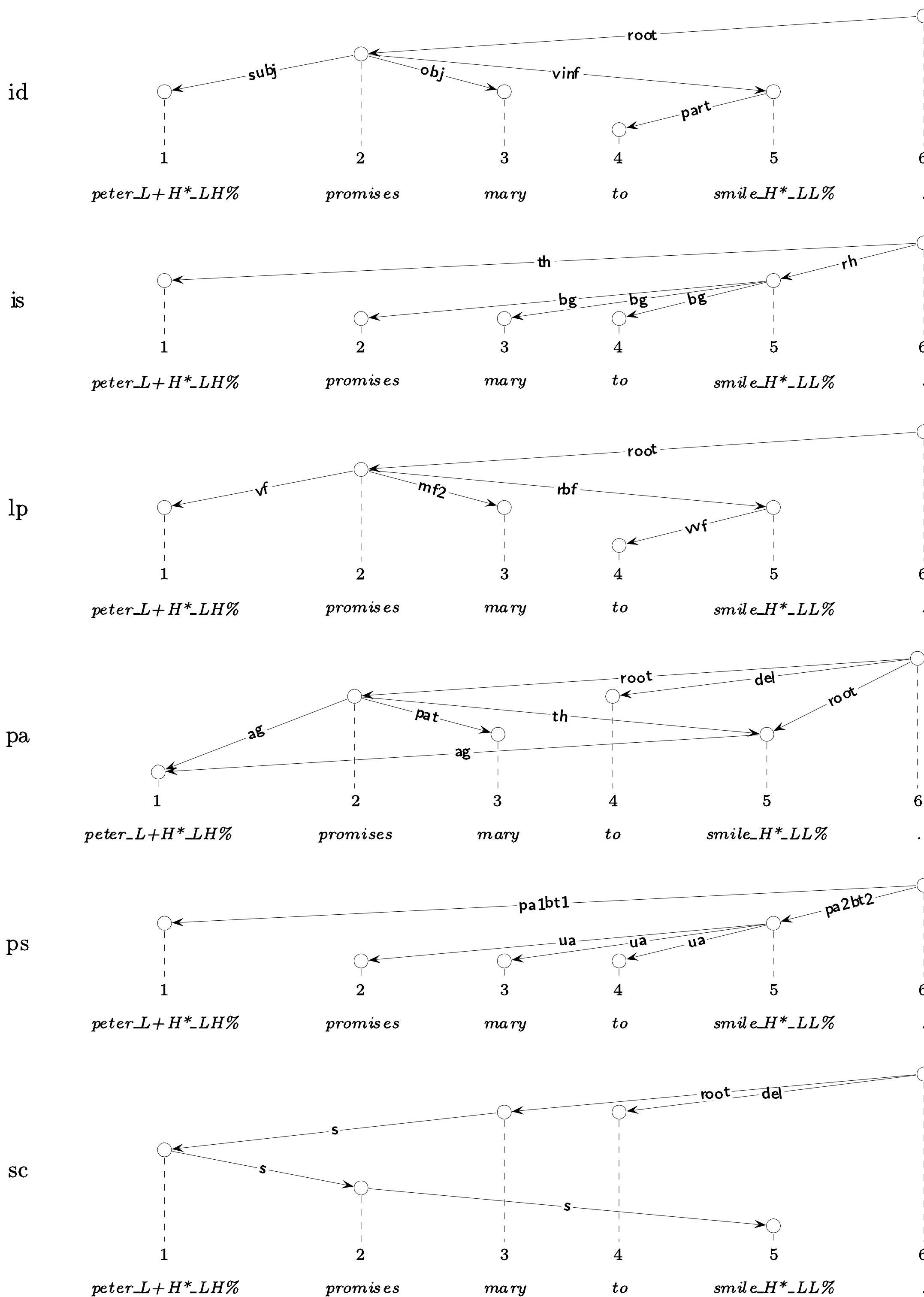

Figura 4: grafo hexadimensional componente de uma análise da frase "Peter promises Mary to smile." (com anotação prosódica de "Peter" e "smile" na sentença dada) segundo a gramática `diss.ul`, distribuída com o XDK 1.7. Diretamente à esquerda de cada projeção posiciona-se o identificador (ID, IS, LP, PA, PS ou SC) da sua respectiva dimensão

O grafo da Figura 4 fornece uma descrição da sentença "Peter promises Mary to smile." nos níveis da sintaxe (dimensões ID e LP), semântica (PA e SC), estrutura prosódica (PS) e estrutura informacional (IS). Por exemplo, podemos observar, em ID, as relações/funções sintáticas usuais, como sujeito (aresta com etiqueta **subj**) e objeto (**obj**); em PA, a identificação de papéis temáticos, como agente (**ag**) e paciente (**pat**); em IS, a divisão do enunciado em tema (**th**, elemento de coesão que recupera algo já conhecido) e rema (**rh**, informação nova) na acepção de Steedman (2000), bem como a oposição entre foco (**rh** ou **th**) e *background* (**bg**).

Nesse grafo, já transparece o apoio à modularidade (e, portanto, instanciabilidade) oferecido pela XDG, aqui concretizado na *separação de interesses*[10] realizada pelas dimensões. Em específico, é importante notar que conceitos comumente mesclados nos modelos de análise usuais estão aqui *desacoplados* — e antes para promover mais sinergia e reusabilidade do que apenas por estética. É o caso da diferenciação entre PA e SC, esta última tratando apenas do escopo das variáveis na representação lógica da sentença, notadamente uma fonte considerável de ambigüidade. Por exemplo, as duas leituras de "*Every man loves a woman.*" diferem exatamente nesse ponto, como se pode observar em (IV.1), em que todo homem ama alguma mulher, e (IV.2), em que uma única mulher é amada por todos os homens.

$$\forall x \left(man(x) \rightarrow \exists y \left(woman(y) \wedge love(x,y) \right) \right) \tag{IV.1}$$

$$\exists y \left(woman(y) \wedge \forall x \left(man(y) \rightarrow love(x,y) \right) \right) \tag{IV.2}$$

---

[10] Tradução aqui adotada do termo clássico *separation of concerns* da Engenharia de Software.

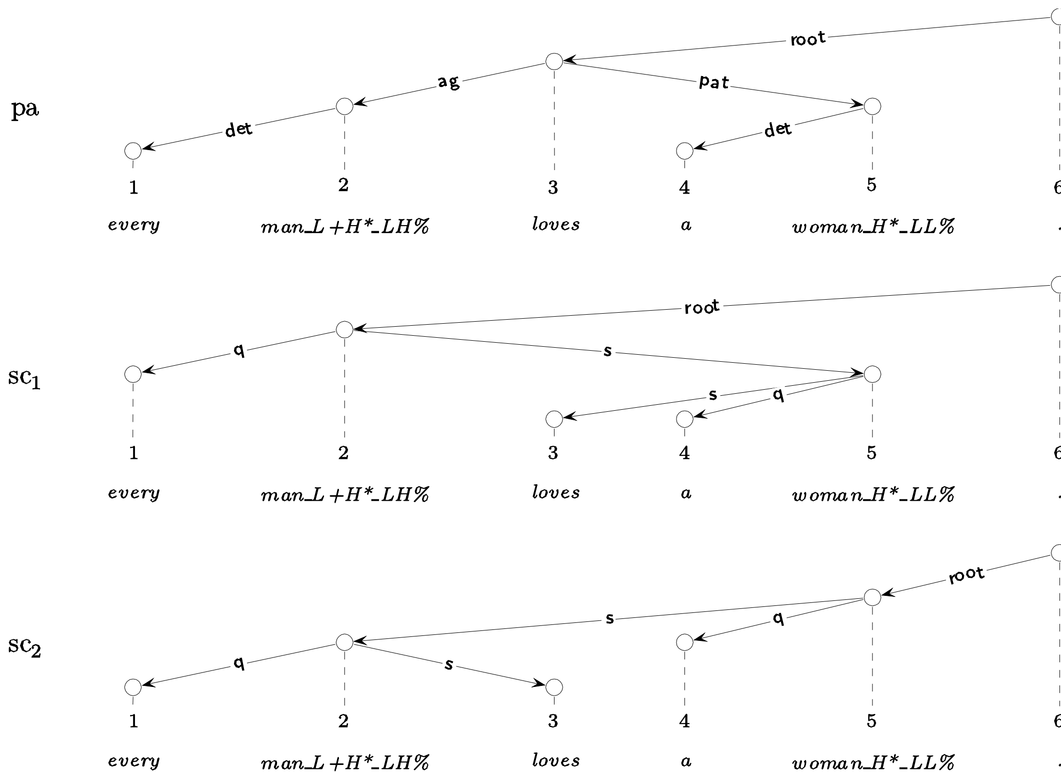

Figura 5: projeções em SC alternativas para as duas análises de "Every man loves a woman." segundo **`diss.ul`**. SC$_1$ e SC$_2$ correspondem respectivamente às equações (IV.1) e (IV.2). As projeções nas demais dimensões (incluindo PA, aqui representada) ficam constantes.

A oposição entre (IV.1) e (IV.2) fica capturada pela gramática **`diss.ul`** como apresentado na Figura 5, enquanto a projeção em PA — bem como nas demais projeções — continua constante entre as duas análises/dois grafos possíveis. Uma grande vantagem dessa modularização é a maior propagação obtida, já que PA, que tem uma interface bastante ativa com ID, pode ficar completamente especificada apesar de indeterminações em SC. Além disso, o XDK pode ser configurado para não gerar análises distintas que difiram apenas em SC (ou qualquer outra dimensão), retornado-as como uma única análise subespecificada.

Outro caso de modularização/separação de interesses crucial ocorre no desdobramento da sintaxe em duas dimensões, ID e LP, capturando respectivamente (i) relações sintáticas e (ii) ordem linear das palavras. A Figura 6 ilustra essa diferenciação, apresentando as projeções em ID e LP de uma análise de "*With whom does Peter laugh?*", também segundo **`diss.ul`**. Con-

forme exemplificado na figura, a árvore em ID não precisa ser **projetiva**, isto é, pode conter subárvores (constituintes) descontínuas(os), como é o caso de "[*with whom*] ... [*laugh*]". Com efeito, essa dimensão é formulada sem qualquer referência à noção de ordem, excetuando-se sua relação com a dimensão LP. *Grosso modo* LP é uma versão possivelmente achatada de ID, ou seja, permite-se, de forma controlada, que algumas subárvores de ID se insiram em LP diretamente abaixo de vértices que as dominem apenas indiretamente em ID. É o que acontece com o constituinte "*with whom*" na Figura 6, o qual surge em LP diretamente abaixo de "*does*", um seu ascendente em ID. Posto de forma figurada, é permitido que algumas subárvores *subam* ou *escalem*. Além disso, LP é ordenada e deve ser necessariamente projetiva. A projetividade é uma restrição típica em Sintaxe e algumas vezes se apresenta forte demais, dificultando a descrição de certos fenômenos lingüísticos, especialmente nas línguas de ordem mais livre, como o alemão. A XDG supera essa limitação abstraindo o conceito e permitindo aplicá-lo de forma explícita e, acima de tudo, *localizada*. É exatamente devido a essa *localização da projetividade* que a gramática **`diss.ul`**, formulada num *framework* não-transformacional, consegue modelar o que em outros sistemas — como a Teoria X-Barra — é tratado como um **movimento**, uma transformação de uma árvore em outra pelo reposicionamento de uma subárvore. De fato, pode-se dizer que o grafo da Figura 6 capture um movimento apesar de nada jamais ter mudado de lugar, o que é possível pela simples supressão da noção de lugar em ID.

Outra restrição típica é a arboreidade, também abstraída/localizada pela XDG. Em específico, **`diss.ul`** não impõe que as projeções em PA sejam árvores, apenas dígrafos acíclicos, o que permite que "*peter*" seja agente de "*promise*" e "*laugh*" a um só tempo na Figura 4. E, dessa forma elegante, pelo mero levantamento de uma restrição para uma dimensão, Debusmann (2006) dispensa o uso de categorias vazias para tratar fenômenos complexos de controle.

Finalizando nossa introdução aos grafos das análises XDG, cumpre destacar dois recursos comuns de modelagem empregados na Figura 4. O primeiro deles é o uso de um vértice artifi-

cial para servir de ponto de partida a todas as projeções, o qual, em **diss.ul**, foi estipulado como o vértice correspondente ao sinal de pontuação (ponto final, de interrogação, etc.) que fecha a frase. Esse vértice é dito **raiz da análise**, mesmo que, em geral, algumas projeções não sejam árvores. O segundo recurso consiste em modelar a **deleção** de vértices em certas dimensões por meio de arestas com um rótulo reservado, usualmente **del**. É o que acontece com "*to*" em PA, já que se trata de uma palavra semanticamente vazia. Normalmente, um vértice a ser considerado deletado numa dimensão qualquer recebe nela uma única aresta, de rótulo **del**, partindo diretamente da raiz da análise.

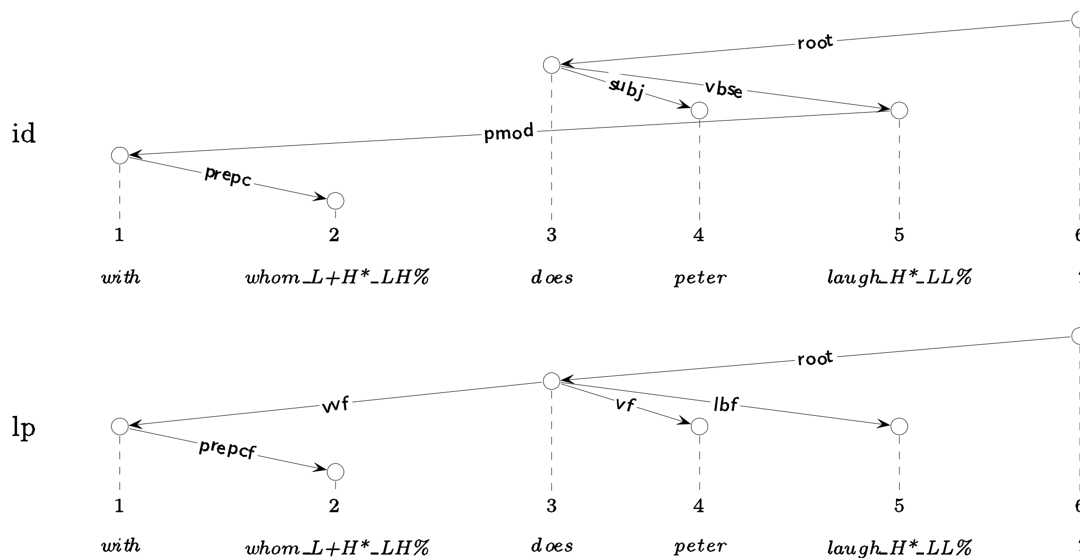

Figura 6: projeções em ID e LP (as duas projeções sintáticas da gramática **diss.ul**) para uma análise de "*With whom does Peter laugh?*"

Atributos

Até o presente momento, tratamos apenas dos grafos XDG e o que eles costumam representar. Entretanto, um grafo é apenas uma parte, mesmo que substancial, de uma análise XDG, que ainda conta com (i) um **mapeamento vértice-palavra *vw*,** que esteve subentendido em nossa discussão até agora e explícito em todas as figuras de grafos apresentadas anteriormente, (ii) uma **ordem total** sobre o conjunto de vértices do grafo, *idem*, e (iii) um **mapeamento vérti-**

ce-atributos, que enfocaremos agora. Tal mapeamento consiste numa função $va : \mathbb{V} \to D \to A \to U$, onde: $\mathbb{V}/D$ é o conjunto de vértices/dimensões do grafo; *A*, um conjunto de átomos identificadores de atributos; e *U*, o universo dos valores que podem ser assumidos pelos atributos. Em outras palavras, cada vértice de uma análise XDG está associado a uma **matriz atributo-valor** (AVM, do inglês *Attribute-Value Matrix*) indexada primeiramente por dimensão, valendo lembrar que os elementos de *U* podem, entre outros valores estruturados, ser também AVMs.

Naturalmente, a presença do mapeamento vértice-atributos é bastante previsível, uma vez que quase todo formalismo gramatical usado na prática conta com um mecanismo análogo. Afinal, uma análise deve, no mínimo, conter uma representação dos traços morfológicos das palavras de uma frase, permitindo-lhe dar conta de fenômenos como concordância, por exemplo. Com efeito, apesar de não estar representado na Figura 4, há o seguinte atributo:

$$va(2)(id)(agrs) = \{(third, sg, masc, nom), (third, sg, fem, nom), ..., (third, sg, neut, acc)\}$$

ou, em linguagem do XDK:

$$va(2)(id)(agrs) = (\$\ third\ \&\ sg),$$

ou seja, o vértice 2, que está associado a "*promises*", tem como valor para o atributo *agrs* da dimensão $\mathbb{ID}$ o conjunto de todas as possíveis tuplas de traços de concordância — pessoa × número × gênero × caso gramatical — tais que sejam da terceira pessoa do singular. Além disso, o mesmo vértice tem o atributo

$$va(2)(id)(agr) = (third, sg, masc, nom),$$

que é a tupla pertencente a $va(2)(id)(agrs)$ efetivamente ativa na frase em questão, permitindo a concordância entre "*Peter*" e "*promises*".

Muitos atributos são **lexicais**, o que significa que a gramática estipula que eles devem provir diretamente de uma entrada lexical existente para a palavra associada ao vértice em questão. No exemplo em vista, o atributo *agrs* da dimensão ID é lexical, de forma que $va(2)(id)(agrs)$ é oriundo de alguma entrada lexical para "*promises*"; *agr*, por sua vez, não o é, mas existe uma prescrição de **`diss.ul`** impondo a seguinte invariante a todas as suas análises:

$$\forall v\big(va(v)(id)(agr) \in va(v)(id)(agrs)\big), \tag{IV.3}$$

de onde se tira que $va(2)(id)(agr)$ deve necessariamente ser uma tupla pertencente a $va(2)(id)(agrs)$.

Não é exagero dizer que os atributos, sobretudo os lexicais, ditam a feição dos grafos das análises XDG. Para citar apenas o exemplo mais eminente, a **valência** de um vértice numa dada dimensão *d* é normalmente um atributo lexical e determina, para cada rótulo de aresta *l* possível em *d*, quantas (zero, uma, zero ou mais, ou uma ou mais) arestas de rótulo *l* o vértice pode receber/emitir em *D*. Outro caso importante abrange os atributos que regulam a correspondência entre papéis temáticos e funções sintáticas. Por exemplo, em **`diss.ul`**, um verbo básico na voz ativa especifica atributos como (IV.4) e (IV.5) abaixo:

$$\begin{aligned} &va(\boldsymbol{v})(idpa)(linkAboveBelow1or2Start)(\boldsymbol{ag}) = \{subj, obj, iobj, pobj1, pobj2\} \\ &va(\boldsymbol{v})(idpa)(linkAboveBelow1or2Start)(x) = \varnothing, \quad x \neq ag \end{aligned} \tag{IV.4}$$

$$\begin{aligned} &va(\boldsymbol{v})(idpa)(linkBelow1or2Start)(\boldsymbol{pat}) = \{\boldsymbol{obj}\} \\ &va(\boldsymbol{v})(idpa)(linkBelow1or2Start)(x) = \varnothing, \quad x \neq pat \end{aligned} \tag{IV.5}$$

Essas equações implicam que existe uma dimensão IDPA (isto é, a interface entre ID e PA), que foi omitida da Figura 4 visto ser necessariamente desprovida de arestas e que surge apenas para agrupar atributos e outros fins administrativos. Encontram-se, em IDPA, os atributos *linkAboveBelow1or2Start* e *linkBelow1or2Start*, cada qual uma função mapeando rótulos de aresta em PA em conjuntos de rótulos em ID. Em primeiro lugar, (IV.4) e (IV.5) regulam que

funções sintáticas um vértice v aceita de acordo com seu papel temático. Caso v seja agente, *linkAboveBelow1or2Start* é não-vazio, e v deve exercer uma das funções presentes no conjunto retornado — sujeito, objeto direto, objeto indireto não-preposicionado (**`iobj`**) ou preposicionado (**`pobj1`** e **`pobj2`**). Caso seja paciente, *linkBelow1or2Start* é não-vazio, exigindo que v assuma apenas a função de objeto direto. Por fim, caso v não seja nem um nem outro, *linkAboveBelow1or2Start* e *linkBelow1or2Start* não têm qualquer efeito.

A diferença entre esses dois atributos está nos requisitos exigidos dos candidatos a pai de v em $\mathbb{ID}$. Dado $u \overset{pat}{\longrightarrow}_{pa} v$, *linkBelow1or2Start* exige $u \overset{obj}{\rightarrow}[\rightarrow]_{id}\, v$, ou seja, v deve se manter abaixo de u. Por outro lado, se $u \overset{ag}{\longrightarrow}_{pa} v$, então *linkAboveBelow1or2Start* impõe um pai w tal que as seguintes condições valham:

$$\begin{array}{l} w \trianglelefteq_{id} u \wedge \\ \exists i \left(\begin{array}{l} w \overset{i}{\rightarrow}[\rightarrow]_{id}\, v \wedge \\ i \in \{subj, obj, iobj, pobj1, pobj2\} \end{array} \right) \end{array}$$

ou seja, em $\mathbb{ID}$, v pode ser filho (i) mesmo de u ou ainda (ii) de um ascendente de u em $\mathbb{ID}$, o que corresponderia a um caso de alçamento ou de controle. A diversidade das possíveis funções sintáticas oferecidas por *linkAboveBelow1or2Start* se deve exatamente ao caso (ii), em que sabemos que a inserção sintática pode variar bastante com as características lexicais de w. No final, a função e o pai correto devem emergir da sinergia entre a restrição associada a *linkAboveBelow1or2Start* e outras, tais como a própria valência.

Analogamente, um particípio passado a ser empregado na construção da voz passiva assume normalmente os valores definidos em (IV.6) e (IV.7). Nesse caso, o rótulo **`pobj2`** denota a função de agente da passiva.

$$\begin{array}{l} va(\boldsymbol{u})(idpa)(linkAboveBelow1or2Start)(\boldsymbol{pat}) = \{subj, obj, iobj, pobj1, pobj2\} \\ va(\boldsymbol{u})(idpa)(linkAboveBelow1or2Start)(x) = \varnothing, \quad x \neq pat \end{array} \qquad \text{(IV.6)}$$

$$va(\boldsymbol{u})(idpa)(linkBelow1or2Start)(\boldsymbol{ag}) = \{\boldsymbol{pobj2}\}$$
$$va(\boldsymbol{u})(idpa)(linkBelow1or2Start)(x) = \varnothing, \quad x \neq ag \qquad \text{(IV.7)}$$

Agora que dispomos de uma boa noção do que seja uma análise XDG, podemos tratar das gramáticas que as originam.

IV.2 Gramáticas XDG

Uma gramática XDG é uma descrição finita e compacta de um conjunto potencialmente infinito de análises XDG. A XDG pode, portanto, ser entendida como uma linguagem muito flexível de escrita desse tipo de gramática, isto é, de descrição de conjuntos de análises. Nesta seção, introduziremos os constructos dessa linguagem e sua realização em uma implementação, o XDK. Na verdade, o XDK disponibiliza três "sistemas de escrita" equivalentes para a XDG, dentre os quais utilizaremos apenas o formato mais adequado à manipulação humana, conhecido como ***User Language*** **(UL)**.

Uma gramática *G* é uma tupla (AT, lex, P) onde *AT* é o tipo das análises de *G*, o que engloba o tipo dos grafos que as compõem e o tipo das AVMs a que os vértices desses grafos devem estar associados; *lex* é o léxico de *G*, consistindo em uma função mapeando cada possível palavra em um conjunto de entradas lexicais alternativas; e *P* é o conjunto de princípios gramaticais de *G*, isto é, restrições que devem ser atendidas por qualquer análise bem-formada de *G*. Embora todos esses elementos sejam passíveis de uma formalização adequada (cf. Debusmann, 2007), ater-nos-emos à intuição por trás deles e à forma como podem ser instanciados no XDK. Em todos os exemplos em UL, usamos excertos do código-fonte de **`diss.ul`**.

Tipo das análises

Uma gramática tem total liberdade para definir quantas dimensões suas análises deverão conter e o que representar em cada uma delas. A definição de uma dimensão *d* abrange basicamente os seguintes itens: (i) o identificador de *d*, que deve ser único na gramática; (ii) o

conjunto de rótulos aceitáveis para as arestas em *d*, (iii) o tipo das AVMs a serem associadas em *d* aos vértices das análises; (iv) a identificação dos atributos lexicais.

```
defdim id {
    … rótulos de aresta …
    … atributos não-lexicais …
    … atributos lexicais …
    … manifesto de grafo …
    … princípios …
    … outros …
}
```

Figura 7: declaração da dimensão ID

O conjunto das definições de dimensão de uma gramática *G* compõe o que se denomina o **tipo das análises** de *G*. Usa-se aqui o termo "tipo" como em "tipo de dados", ou seja, (uma especificação de) um conjunto de valores. Em UL, a definição de uma dimensão identificada como ID toma a forma apresentada na Figura 7. Vale observar que as diferentes seções dentro de uma tal declaração podem aparecer em qualquer ordem e que apenas as partes sombreadas concernem ao tipo das análises.

A Figura 8 apresenta a definição dos rótulos de aresta em ID. Em primeiro lugar, define-se, nas linhas 1 a 3, um tipo enumerado — **`id.label`** — abrangendo todos os valores pretendidos para os rótulos. Na listagem, as aspas são usadas para construir um identificador contendo um ou mais caracteres não-alfanuméricos, como é o caso de **`id.label`**, cuja natureza é, de resto, idêntica à de qualquer outro identificador, como **`id`** ou **`adj`**. Até esse ponto, tudo o que se fez foi definir um tipo auxiliar global, que pode inclusive ser usado na definição de outras dimensões, e lhe dar um nome sugestivo. É justamente por meio da diretiva **`deflabeltype`**, na linha 5, que **`id.label`** se torna o tipo dos rótulos de aresta de ID. O argumento de **`deflabeltype`** deve necessariamente ser um tipo enumerado.

```
  defdim id {
1    deftype "id.label" {adj adv comp det iobj obj
2       part pmod pobj1 pobj2 prepc rel root sub
3       subj vbse vinf vprt}
4
5    deflabeltype "id.label"
```

```
    … atributos não-lexicais …
    … atributos lexicais  …
    … manifesto de grafo …
    … princípios …
    … outros …
}
```

Figura 8: definição dos rótulos das arestas em ID

O tipo das AVMs a serem associadas aos vértices numa dimensão é definido, no XDK, como dois tipos de AVM, um para os atributos lexicais e outro para os demais. A Figura 9 apresenta a definição deste último em ID, incluindo tipos auxiliares. Desta vez, são definidos diversos tipos enumerados (linhas 1 a 7), todos modelando traços morfológicos usuais exceto por **`id.pagr`**, que é utilizado para controlar fenômenos de regência/seleção de preposições. A linha 5 é digna de nota por conter o construtor **`tuple`**, que cria tipos de tuplas a partir de tipos mais simples; **`id.agr`** é, portanto, o tipo dos feixes de traços morfológicos envolvidos em fenômenos de concordância. Feitas essas diversas declarações auxiliares, os atributos não-lexicais de ID são definidos nas linhas 9 e 10, por meio da diretiva **`defattrstype`**. Trata-se de um tipo de AVM com dois atributos: **`agr`**, que já usamos como ilustração na Seção IV.1 (pág. 59), e **`pagr`**.

```
   defdim id {
       … rótulos de aresta …
1      deftype "id.person" {first second third}
2      deftype "id.number" {sg pl}
3      deftype "id.gender" {masc fem neut}
4      deftype "id.case" {nom acc}
5      deftype "id.agr" tuple("id.person" "id.number"
6         "id.gender" "id.case")
7      deftype "id.pagr" {at by in of on to with}
8
9      defattrstype {agr: "id.agr"
10                   pagr: "id.pagr"}
       … atributos lexicais  …
       … manifesto de grafo …
       … princípios …
       … outros …
   }
```

Figura 9: definição dos atributos não-lexicais de ID e tipos auxiliares

```
   defdim id {
       … rótulos de aresta …
       … atributos não-lexicais …
```

```
1      defentrytype {in: valency("id.label")
2                     out: valency("id.label")
3                     agrs: iset("id.agr")
4                     pagrs: iset("id.pagr")
5                     pobj1: iset("id.pagr")
6                     pobj2: iset("id.pagr")}
       … manifesto de grafo …
       … princípios …
       … outros …
   }
```

Figura 10: definição dos atributos lexicais em ID. Utilizam-se aqui os tipos auxiliares definidos na Figura 8 e na Figura 9.

A definição dos atributos lexicais de uma dimensão é realizada de forma idêntica, salvo pelo uso da diretiva **defentrytype** em vez de **defattrstype**, como mostra a Figura 10. Nela, são especialmente dignos de nota os construtores **valency** e **iset**. O primeiro constrói um tipo para valências dado um tipo de rótulos de aresta; e é com ele que se definem os atributos **in** e **out**, presentes em praticamente todas as dimensões de qualquer gramática XDG e que regulam respectivamente as valências de entrada e saída de cada vértice. Dado um tipo T, o construtor **iset**, por sua vez, cria 2^T, ou seja, o tipo dos conjuntos cujos elementos pertencem a T. Assim, o atributo lexical **agrs** de uma palavra aceita como valor um conjunto de feixes de traços de concordância — na verdade, todos os possíveis para a palavra em questão, fato que já usamos como ilustração na Seção IV.1 (pág. 59).

```
   defdim id {
       … rótulos de aresta …
       … atributos não-lexicais …
       … atributos lexicais …
1      useprinciple "principle.graphConstraints" {
2         dims {D: id}
3      }
4      useprinciple "principle.graphDist" { dims {D: id} }
       … princípios …
       … outros …
   }
```

Figura 11: manifesto de que ID é, em geral, uma projeção não-vazia

Em teoria, o que declaramos até o momento deveria ser suficiente para criar uma dimensão numa gramática XDG. Entretanto, o XDK pode exigir que ainda façamos duas declarações,

como mostra a Figura 11. Ambas se fazem por meio da aplicação de princípios, que analisaremos em maior detalhe mais adiante. Por ora, basta saber que, caso esperemos que uma dimensão de fato venha a conter arestas, é preciso aplicar-lhe o princípio **`graphConstraints`** (linhas 1 a 3). Isso ocorre para evitar o desperdício de recursos computacionais, pois é costumeiro definir diversas dimensões não-gráficas (como IDPA, mencionada anteriormente) numa gramática, apenas para organizar atributos e facilitar a aplicação de princípios. Naturalmente, a aplicação de **`graphConstraints`** a tais dimensões não mudaria de forma alguma o conjunto de análises geradas, seria mesmo apenas um desperdício. A segunda declaração, necessária para permitir um maior controle da saída de acordo com os requisitos da aplicação, diz respeito a se desejamos necessariamente análises completamente especificadas para a dimensão em questão. Em caso afirmativo, faz-se necessário aplicar também o princípio **`graphDist`**.

Princípios gramaticais

O tipo das análises de uma gramática pode ser considerado uma restrição importante sobre a feição das mesmas, porém sempre será por demais irrestrito para ter qualquer significado por si só. Em específico, ficasse uma gramática definida apenas pelo tipo de declarações que vimos até o momento, não haveria qualquer relação, por exemplo, entre os atributos lexicais **`in`** e **`out`** e as valências atuais dos vértices. Na verdade, o conceito de valência nem sequer existiria, e qualquer aresta de qualquer rótulo seria válida entre qualquer par de vértices. Em geral, nenhum atributo — lexical ou não — teria qualquer efeito sobre a construção das análises. Não existiria nenhum tipo de interdependência entre as dimensões. Em resumo, todo tipo de restrição adicional — tal como a pertinência do atributo **`agr`** ao conjunto **`agrs`**, o uso de **`agr`** para estabelecer concordância ou ainda a influência exercida pelos atributos **`linkAboveBelow1or2Start`** e **`linkBelow1or2Start`** sobre a realização sintática de papéis temáticos — precisa ser prescrito explicitamente pela gramática. Eis a acepção primeira do termo “princípio”:

princípio₁: qualquer restrição adicional imposta por uma gramática sobre o tipo de suas análises.

Eis também uma das grandes tarefas do engenheiro gramatical XDG: aplicar um conjunto de princípios que elimine, dentre todos os valores pertencentes ao tipo das análises, todas e somente as análises malformadas. Formalmente, todo princípio é uma fórmula em lógica de primeira ordem envolvendo quantificação sobre os conjuntos de vértices, arestas e dimensões de uma análise e recorrendo a um repertório bem limitado de predicados. Trata-se de uma linguagem muito simples e clara que introduziremos mais adiante, quando trataremos do *Principle Writer* (pág. 79).

Importa neste momento a segunda acepção do termo, qual seja:

princípio₂: uma abstração de um princípio₁, ou seja, um princípio₁ parametrizado e, portanto, reusável, podendo ser aplicado diversas vezes numa mesma gramática (gerando, assim, diversos princípios₁ concretos) ou ainda em gramáticas diferentes.

O XDK oferece uma ampla biblioteca de princípios₂, que pode ser facilmente estendida por meio do *Principle Writer* ou, nem tão facilmente, pela programação de módulos especiais em Oz.

```
     defdim id {
        … rótulos de aresta …
        … atributos não-lexicais …
        … atributos lexicais  …
        … manifesto de grafo …
1       useprinciple "principle.tree" { dims {D: id} }
2       useprinciple "principle.valency" {
3         dims {D: id}
4         args {In: _.D.entry.in
5               Out: _.D.entry.out}}
6       useprinciple "principle.agr" {
7         dims {D: id}
8         args {Agr: _.D.attrs.agr
9               Agrs: _.D.entry.agrs}}
10      useprinciple "principle.agreement" {
11        dims {D: id}
12        args {Agr1: ^.D.attrs.agr
```

```
13              Agr2: _.D.attrs.agr
14              Agree: {det subj}}}
          … outros …
   }
```

Figura 12: aplicação de princípios em ID

A Figura 12 exemplifica a aplicação de alguns princípios da biblioteca padrão do XDK 1.7. Na linha 1, o primeiro deles — **`tree`** — impõe arboreidade às projeções em ID. Trata-se de um princípio com um único parâmetro, a dimensão a ser restrita. Em seguida, nas linhas 2 a 5, aplica-se o princípio **`valency`**, que instaura a noção de valência. Note que, além da dimensão a ser restrita, esse princípio tem dois parâmetros adicionais, **`In`** e **`Out`**, que indicam a fonte das valências de entrada e saída dos vértices. É justamente ao se passarem os argumentos **`_.D.entry.in`** e **`_.D.entry.out`** que as valências em ID passam a ser lexicalizadas, provindo dos atributos lexicais **`in`** e **`out`**. Esses argumentos são dados como expressões de acesso no formato

```
<vértice>.<dimensão>.(entry|attrs).<atributo>
```

e devem ser interpretados da seguinte forma: "**`_`**" é uma variável que representa cada vértice de uma análise numa iteração implícita; **`D`** é o próprio parâmetro dimensional do princípio e, nesse momento, vale ID; e **`entry`**/**`attrs`** fornece acesso aos atributos lexicais/ordinários da dimensão que o precede na expressão. A já comentada relação de pertinência entre os atributos **`agr`** e **`agrs`** é imposta pelo princípio homônimo **`agr`** (linhas 6 a 9). Entretanto, a relação de concordância propriamente dita só é estabelecida por **`agreement`** (linhas 10 a 14). Trata-se de um princípio que difere dos anteriores por iterar por arestas, não vértices. Nesses casos, surge a variável "**`^`**", representando a origem da aresta corrente, enquanto "**`_`**" passa a denotar o seu destino. Essa aplicação de **`agreement`** estipula que, para toda aresta *e* em ID, se o rótulo de *e* for **`subj`** ou **`det`**, então os atributos **`agr`** dos vértices ligados por *e* devem ser iguais. Em outras palavras, o princípio assegura que haja concordância entre todo verbo e seu sujeito e entre todo determinante e seu determinado.

```
   defdim idpa {
1     defentrytype {
2        ...
3        linkBelow1or2Start: vec("pa.label" set("id.label"))
4        linkAboveBelow1or2Start: vec("pa.label" set("id.label"))
5        ...
6     }
7
8     useprinciple "principle.linkingBelow1or2Start" {
9        dims {D1: pa
10             D2: id
11             D3: idpa}
12       args {Start: ^.D3.entry.linkBelow1or2Start}}
13    useprinciple "principle.linkingAboveBelow1or2Start" {
14       dims {D1: pa
15             D2: id
16             D3: idpa}
17       args {Start: ^.D3.entry.linkAboveBelow1or2Start}}
      ...
   }
```

Figura 13: princípios multidimensionais em IDPA

Nos exemplos apresentados, figuram apenas princípios unidimensionais. Há, contudo, aqueles parametrizados para mais de uma dimensão, estabelecendo algum tipo de dependência ou comunicação interdimensional. É o caso dos princípios **`linkingAboveBelow1or2Start`** e **`linkingBelow1or2Start`** que "dão vida" respectivamente aos atributos **`linkAboveBelow1or2Start`** e **`linkBelow1or2Start`**, discutidos anteriormente (pág. 60). A Figura 13 demonstra como esses princípios foram aplicados na dimensão IDPA de **`diss.ul`**. Nesse exemplo, vale ainda atentar para a definição dos dois atributos lexicais em questão, em que se faz uso do construtor de funções **`vec`** — no caso, mapeando rótulos de aresta de PA em conjuntos de rótulos de ID.

Léxico

Uma vez declaradas todas as dimensões de uma gramática, o XDK exige ainda que a dimensão LEX seja definida contendo ao menos o atributo lexical **`word`**, do tipo **`string`**, como mostra a Figura 14. Trata-se do atributo que indexará todo o léxico e contra o qual serão comparadas as palavras da entrada. Com isso, completa-se a rotina de definição do tipo das entradas lexicais da gramática, que fica implícito na declaração do tipo das análises. Trata-se

da agregação de todos os tipos declarados **`defentrytype`** na gramática. Efetivamente, o léxico XDG é uma estrutura de dados homogênea, cada uma de cujas entradas é composta unicamente pela totalidade dos atributos lexicais definidos para as dimensões, como ilustrado na Figura 15.

```
defdim lex {
   defentrytype {word: string}
}
```

Figura 14: definição mínima da dimensão LEX

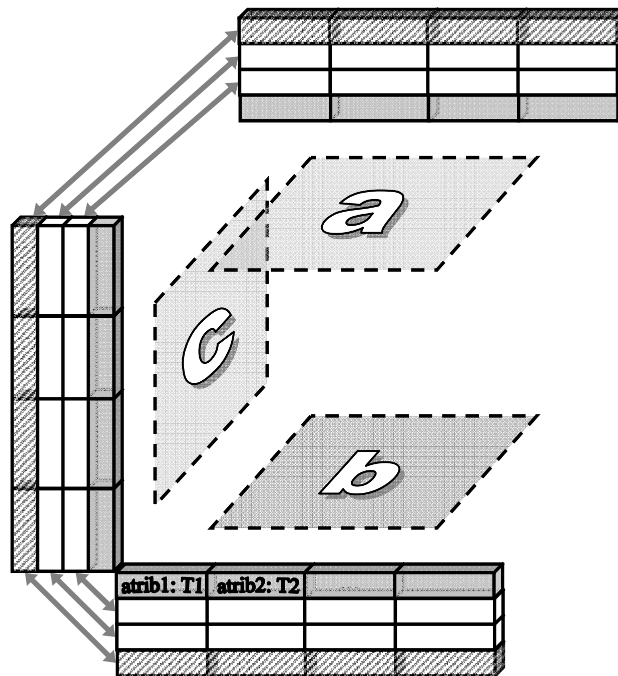

Figura 15: o léxico numa gramática XDG com três dimensões *a*, *b* e *c*. A área hachurada corresponde a uma única entrada lexical, composta pela simples agregação dos atributos lexicais de todas as dimensões

O conjunto de diretivas **`defdim`** de uma gramática define, portanto, o tipo de suas análises, seu conjunto de princípios e o tipo de suas entradas lexicais. No entanto, o seu léxico só será preenchido por meio de diretivas **`defentry`**, cada qual inserindo uma ou mais entradas lexicais. A Figura 16 apresenta parte de uma entrada lexical para a palavra "*promises*" compatível com o léxico de **`diss.ul`**, cobrindo a íntegra das dimensões LEX, ID e IDPA. Em primeiro lugar, vale notar que os atributos lexicais **`pagrs`**, **`pobj1`** e **`pobj2`**, declarados por ID, não são

aí mencionados, o que não representa problema, já que valores *default* são assumidos de acordo com os respectivos tipos dos atributos. Dignos de nota também são os diversos construtores de valor utilizados. O atributo **`agrs`** utiliza uma expressão geradora de conjuntos, descrevendo, de forma sucinta, todos os feixes de concordância que sejam da terceira pessoa do singular. Os atributos **`in`** e **`out`** usam o construtor de valência padrão e, em resumo, expressam que "*promises*" aceita receber até uma aresta **`root`** e requer exatamente um sujeito, um objeto direto e um infinitivo e qualquer número de modificadores (**`adv`** e **`pmod`**), inclusive zero. A valência *default* é zero, ou seja, rótulos de aresta não mencionados não são aceitos. Os atributos **`link*`** de IDPA usam construtores de funções, que, no fundo, não passam de AVMs. Mais uma vez, imagens *default* — no caso, conjuntos vazios — são assumidas para valores não mencionados do domínio de tais funções.

```
   defentry {
1     dim lex { word: "promises" }
2     dim id {
3        agrs: ($ third & sg)
4        in: {root?}
5        out: {subj! obj! vinf! adv* pmod*}
6     }
7     dim idpa {
8        linkAboveBelow1or2Start: {
9           ag: {subj obj iobj pobj1 pobj2}}
10       linkBelow1or2Start: {
11          pat: {obj}}
12       linkBelowStart: {
13          th: {vinf}}
14       lockDaughters: {iobj obj}
15    }
      ...
   }
```

Figura 16: parte de uma entrada lexical para "*promises*" compatível com os dados lexicais de `diss.ul`.

Felizmente, o código-fonte de **`diss.ul`** não contém, para "*promises*", nada parecido com o que mostra a Figura 16. Essa entrada lexical tem muito em comum com as de diversos outros verbos, como as próprias flexões "*promise*" e "*promised*", o infinitivo "*to promise*", outros verbos de controle e mesmo outras flexões da terceira pessoa do singular do presente de outros verbos quaisquer. Em geral, existem tantas oportunidades de reuso na construção de um

léxico que seria uma grave falta quanto à instanciabilidade se o XDK não oferecesse dispositivos para aproveitá-las. E, de fato, eles existem e se concertam em torno de uma abstração básica, uma unidade central de reusabilidade lexical, qual seja o conceito de classe lexical parametrizada. Uma **classe lexical** pode ser entendida como uma entrada lexical parcial, agora introduzida pela diretiva **`defclass`**, que (i) está isenta da obrigação de definir o atributo LEX**`.word`**, (ii) deve, em contrapartida, ter um identificador único entre as demais classes lexicais de sua gramática, (iii) pode declarar um ou mais parâmetros e, por meio de seu identificador e a fixação desses parâmetros, (iv) pode ser aplicada, possivelmente em combinação com outras classes, na construção de diversas entradas lexicais ou mesmo novas classes. Trata-se de um constructo simples, mas poderoso, em grande parte análogo à noção de função das linguagens de programação.

```
1  defclass "id_noun" {
2     dim id {in: {subj? obj? iobj? prepc?}}}
3  ...
4  defclass "id_perpro" A {
5     "id_noun"
6     dim id {agrs: A
7             out: {adj* pmod* rel?}}}
8  ...
9  defclass "perpro" W A {
10    "id_perpro" {A: A}
11    "lp_perpro"
12    "idlp_perpro"
13    "pa_perpro"
14    "idpa_perpro"
15    "sc_perpro"
16    "pasc_perpro"
17    "clls_perpro" {A: W}
18    "lex_enter" {W: W}}
19 ...
20 defentry {
21    "perpro" {W: "i"
22              A: ($ first & sg & nom)}}
23 ...
24 defentry {
25    "perpro" {W: "her"
26              A: ($ fem & third & sg & acc)}}
```

Figura 17: uso de classes lexicais num excerto (descontínuo) de `diss.ul` relativo à definição de "*I*" e "*her*"

Antes de apresentar como "*promise*" é de fato definido em **diss.ul**, analisaremos um caso mais simples, ilustrado pela Figura 17. Trata-se de um excerto descontínuo da gramática, em que apenas as aplicações sombreadas de classes lexicais têm a definição correspondente inclusa, acima no código-fonte. Nas linhas 9 a 18, vemos a definição da classe **perpro**, de pronomes pessoais, que declara os parâmetros **W** — a própria palavra a ser definida — e **A** — seu conjunto de feixes de concordância. Note-se que, na construção de **prepro**, entram aplicações de nove outras classes lexicais, dentre as quais incluímos apenas a definição de **id_prepro**. A essa classe é delegado tudo o que deve ser definido em ID para um pronome pessoal; e, em sua aplicação da linha 10, podemos observar o repasse do parâmetro **A** de **perpro** como o parâmetro homônimo de **id_perpro**. Nas linhas 4 a 7, **id_perpro** usa seu parâmetro para fixar o atributo ID**.agrs**, define a valência de saída em ID e aplica uma classe sem parâmetros, **id_noun**. Esta última fatora o que existe de comum, em ID, a todas as categorias nominais da gramática, o que, no caso, resume-se às funções sintáticas que podem assumir. Por fim, observamos a aplicação de **perpro** na definição dos pronomes "*I*" (linhas 20 a 22) e "*her*" (24 a 26).

```
1  defclass "id_non3sg" {
2     dim id {agrs: ($ (first|second) & sg) |
3             agrs: ($ pl)}}
4  ...
5  defclass "lex_fin2" W3sg Wnon3sg {
6     "lex_fin"
7     ("id_3sg" &
8     "lex_enter" {W: W3sg}) |
9     ("id_non3sg" &
10    "lex_enter" {W: Wnon3sg})}
11 ...
12 defclass "verb" W3sg Wnon3sg Wprt {
13    "lex_fin2" {W3sg: W3sg
14                Wnon3sg: Wnon3sg} |
15    "lex_infin" {Winf: Wnon3sg
16                 Wprt: Wprt}
17    dim clls {anchor: Wnon3sg}}
18 ...
19 defentry {
20    "verb" {W3sg: "promises"
21            Wnon3sg: "promise"
22            Wprt: "promised"}
23    "trsubjsubjctrl"}
```

Figura 18: disjunção e conjunção em classes lexicais num excerto (descontínuo) de `diss.ul` relativo à definição de "*promises*"

Finalmente, podemos tratar da definição de "*promises*", ilustrada pela Figura 18 com um excerto semelhante ao da Figura 17. Essa definição utiliza outro recurso poderoso oferecido pelo XDK para minimizar redundâncias na definição do léxico e que casa à perfeição com classes lexicais, qual seja a **disjunção lexical**, denotada pelo operador "`|`". A intuição por trás dessa operação é muito simples e está mais aparente na definição da classe **`verb`**, nas linhas 12 a 17, que captura o que há de comum a todos os verbos não-auxiliares no presente (simples), infinitivo e particípio (passado)[11]. Em específico, **`verb`** implementa a idéia de que, em inglês, todas as formas de um tal verbo serão conhecidas dada sua flexão da terceira pessoa do singular do presente (parâmetro **`W3sg`**), a flexão única das demais pessoas do mesmo tempo (**`Wnon3sg`**, que coincide com o infinitivo), e o particípio (**`Wprt`**). Basicamente, o que essa definição diz é que um verbo é uma forma finita (classe **`lex_fin2`**) ou não (**`lex_infin`**) e que, em todo caso, o atributo CLLS**`.anchor`** deve ser igual ao infinitivo. De posse de seus parâmetros, **`lex_fin2`** e **`lex_infin`** se encarregam de gerar tudo o que é necessário para o presente e as formas não-finitas, respectivamente.

O que talvez ainda não esteja tão claro é o que exatamente ocorre quando se usa a disjunção. Com ela, nossas classes lexicais deixam de ser meros pedaços reusáveis de entrada lexical e se tornam verdadeiras fábricas automáticas de entradas. Em conseqüência, a diretiva **`defentry`** nas linhas 12 a 23 não corresponde à definição de uma única entrada lexical. No mínimo, é de se esperar que ela desencadeie a produção de três entradas, uma para cada uma das formas "*promises*", "*promise*" e "*promised*"; mas, na verdade, essa única **`defentry`** é responsável pela geração de um total de 252 entradas lexicais, não só para as formas passadas

---

[11] A gramática **`diss.ul`** não trata as demais formas verbais simples, como o passado simples ou o particípio presente.

por parâmetro, mas também para versões anotadas prosodicamente como "*promises_L+H**" e "*promises_LL%*", todas geradas automaticamente por meio de disjunção pela classe **`lex_enter`** (linhas 8 e 10).

Para apreciar o efeito exato da disjunção, é necessário entender o processo desencadeado por uma **`defentry`** como um *pipeline* de transformações/funções sobre multiconjuntos de entradas lexicais. O multiconjunto inicial contém uma única entrada, apenas com valores *default*. Cada (aplicação de) classe lexical corresponde a uma função f tal que $f(S)$ retorna as entradas lexicais de S devidamente atualizadas, também como um multiconjunto. Seguindo esse esquema, a disjunção de classes lexicais gera uma nova transformação que pode ser definida assim:

$$\left(f \mid g\right)(S) = f(S) \cup g(S)\,. \tag{IV.8}$$

Por conseguinte, cada disjunção dobra o número de entradas lexicais do multiconjunto a que foi aplicada, razão pela qual é um recurso que deve ser usado com cautela. Voltando à Figura 18, vale notar, na definição da classe **`lex_fin2`** (linhas 5 a 10), uma aplicação mais complexa da disjunção. Ali vemos o uso da operação de **conjunção**, denotada por "**`&`**". Na verdade, já vimos fazendo uso implícito da mesma operação, o que acontece, por exemplo, na diretiva **`defentry`** (linhas 19 a 23), quando justapomos as aplicações das classes **`verb`** e **`trsubj-subjctrl`**, e em toda a definição de **`perpro`** (linhas 9 a 18 da Figura 17). Retomando nossa interpretação de cadeia de transformações, trata-se de uma composição simples, como definido em (IV.9), que precisa ser explicitada por ocorrer dentro dos operandos de uma disjunção. A equação (IV.10) apresenta uma formulação funcional do significado da classe **`lex_fin2`**.

$$f \,\&\, g = f \circ g \quad \Leftrightarrow \quad \left(f \,\&\, g\right)(S) = f\left(g(S)\right) \tag{IV.9}$$

$$lex_fin2 = lex_fin \circ \left(\left(id_3sg \circ lex_enter_{\{W:W3sg\}} \right) \middle| \left(id_non3sg \circ lex_enter_{\{W:Wnon3sg\}} \right) \right) \tag{IV.10}$$

Finalizando nossas considerações sobre a disjunção, chamamos a atenção para a definição da classe **id_non3sg**, nas linhas 1 a 3 da Figura 18. Em primeiro lugar, lá se encontra um e-xemplo pouco usual de disjunção lexical, denotada pelo operador ao final da linha 2, cujos operandos são duas especificações parciais de AVM. Isso demonstra que praticamente qualquer trecho de especificação lexical pode se tornar um operando. Em segundo lugar, vale notar que o operador “**|**” empregado na expressão **($ (first|second) & sg)**, ainda na linha 2, *não* denota disjunção lexical e, portanto, não tem qualquer efeito multiplicativo sobre o número de entradas lexicais. Assim é por se tratar de uma expressão geradora de conjuntos, sempre introduzida pelo operador “**$**”. Na verdade, a mesma classe poderia ter sido definida de maneira mais eficiente eliminando-se a disjunção lexical, como apresentado na Figura 19.

```
defclass "id_non3sg" {
   dim id {agrs: ($ ((first|second) & sg) | pl) }
}
```

Figura 19: definição mais eficiente da classe `id_non3sg`, alternativa à presente na Figura 18

Satisfação e sincronização lexical

Uma vez compreendida a estrutura do léxico XDG e sua construção no XDK, cumpre atentar para sua função numa gramática. Já introduzimos a idéia de que, como fonte dos atributos lexicais, o léxico tem forte impacto sobre as análises. Em sua formulação geral, tal impacto é conhecido como **satisfação lexical**. Trata-se de um princípio implícito do modelo XDG que diz que, para que uma análise seja bem-formada segundo uma gramática G, não basta atender a todos os princípios gramaticais, mas também é preciso, para cada vértice v da análise, que exista uma entrada lexical e para a palavra de v tal que, para toda dimensão d de G e todo atributo lexical a definido em d, o valor de a para o vértice seja idêntico ao valor de a para e, ou seja, $va(v)(d)(a) = e(d)(a)$. Em outras palavras, para cada vértice v, deve existir uma entrada lexical que, servindo como fonte única de atributos lexicais para v, satisfaça a todos os princípios ao mesmo tempo, em todas as dimensões. Conseqüentemente, a satisfação lexical acaba

implicando um mapeamento vértice-entrada lexical subjacente a qualquer análise bem-formada, como representado na Figura 20.

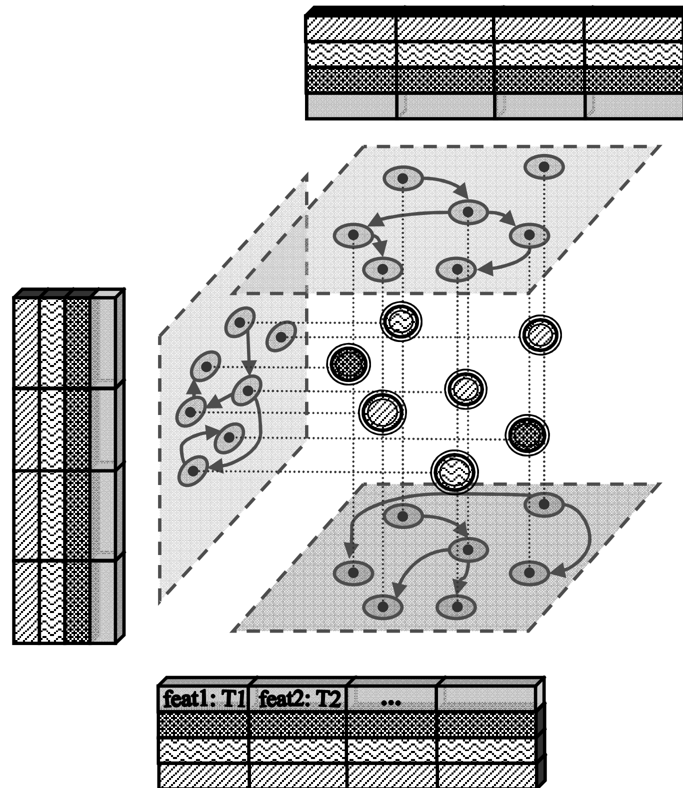

Figura 20: a satisfação lexical de uma análise XDG implica um mapeamento de vértices em entradas lexicais, aqui representado pelo padrão de preenchimento dessas entidades

A satisfação lexical ocasiona um fenômeno conhecido como **sincronização lexical**, que favorece significativamente a propagação de restrições numa implementação por PCR. Trata-se de correlações entre os valores aplicáveis aos diversos atributos lexicais de cada vértice *v* que acabam *emergindo*, ao longo do processamento, localmente ao subconjunto $curlex(v,t) \subseteq lex(vw(v))$ de entradas lexicais aplicáveis a *v* no instante *t*. Isso ocorre porque $curlex(v,t)$, mesmo quando igual ao conjunto total de entradas $lex(vw(v)) = curlex(v,t_0)$ para

a palavra $vw(v)$, é extremamente reduzido em relação ao tipo das entradas lexicais[12]. A Figura 21 dá um exemplo hipotético de uma tal situação, em que se sabe que $\{e_1, e_2, e_3\} \subseteq lex(vw(v))$ é o conjunto de entradas lexicais correntemente aplicáveis a um dado vértice *v*. Na figura, enfocam-se os atributos *A*, da dimensão D_1, e *B*, de D_2, e as correlações que surgem entre seus possíveis valores. Seja por distribuição ou propagação promovida pelos princípios gramaticais, novas restrições deverão ser impostas a qualquer momento. Por exemplo, se a restrição $va(v)(D_2)(B) = \boxplus$ for imposta, disso decorrerá que $va(v)(D_1)(A) = \circledcirc$ e $curlex(v,t) = \{e_1\}$. Mais interessante ainda seria o que aconteceria caso se impusesse a restrição $va(v)(D_1)(A) \neq \circledcirc$, o que impediria a seleção de e_1. Nesse caso, ainda que o atributo $va(v)(D_1)(A)$ continuasse indeterminado, teríamos, por propagação, a total determinação de $va(v)(D_2)(B)$, que assumiria o valor $\boxdot$, o único possível para $curlex(v,t) = \{e_2, e_3\}$. Tais novas restrições, causadas meramente pela sincronização lexical, possivelmente promoverão propagação adicional. Considerando ainda que tudo isso esteja valendo para todos os atributos de todos os vértices ao mesmo tempo, podemos pressentir o potencial desse mecanismo no controle da complexidade quando se aplica PCR à busca por análises XDG.

---

[12] O tipo das entradas lexicais é o conjunto de todas as entradas lexicais definíveis, mesmo as que não constem do léxico.

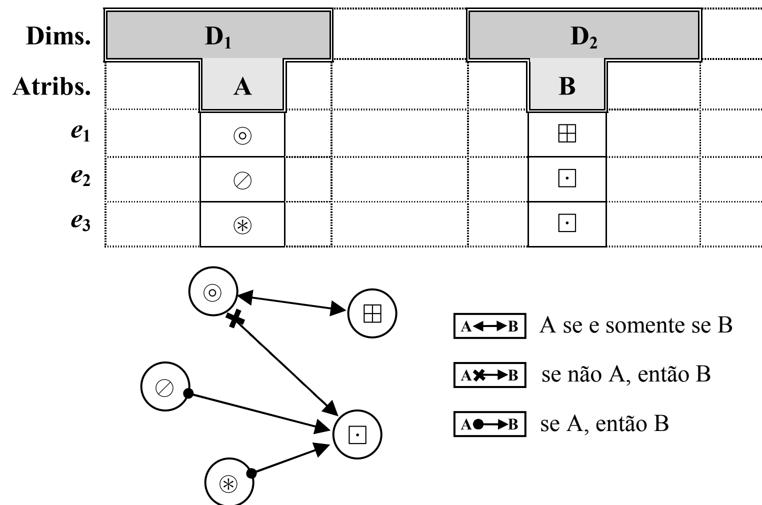

Figura 21: conjunto $\{e_1, e_2, e_3\}$ de possíveis entradas lexicais para um vértice *v* e algumas correlações entre valores de atributo num dado instante da geração de uma análise XDG

IV.3 Principle Writer

Apesar de a biblioteca de princípios parametrizados do XDK ser ampla e bastante genérica, extensões acabam por se fazer necessárias em praticamente toda nova aplicação/gramática, o que foi especialmente verdadeiro ao tentarmos fazer geração com um sistema que, até então, só tinha sido usado para *parsing*. Originalmente, criar novos princípios$_2$ não era tarefa fácil. Em específico, isso implicava programar numa linguagem de uso geral não muito conhecida — Oz — manipulando uma API razoavelmente complexa, um trabalho que, muitas vezes, requeria qualificações ausentes no engenheiro gramatical, exigindo a intervenção de outro especialista. Em qualquer caso, tratava-se do grande ponto fraco do XDK, no quesito instanciabilidade, que foi completamente sanado com o advento do ***Principle Writer*** **(PW)**, incluso no XDK a partir da versão 1.6.33. O PW é um compilador para uma linguagem de definição de princípios — a ***Principle Language*** (**PL**) — compacta, simples e acessível a qualquer usuário que domine os conceitos da XDG.

A Tabela 2 elenca os principais constructos da PL e seus respectivos significados. Simples que pareça, essa linguagem é capaz de expressar todos os princípios que podem ser impostos

por meio da biblioteca do XDK. Para ilustrar o uso da PL, apresentamos, na Figura 22, uma versão do princípio **`agreement`**, a qual integra a distribuição do XDK.

Tabela 2: principais constructos da PL e seus respectivos significados. A letra ϕ representa subexpressões lógicas; ε, subexpressões não-lógicas; τ, expressões de tipo; e as demais, identificadores de variável dos seguintes tipos: dimensão (d), rótulo de aresta (l), vértice (u e v) e qualquer (x e y)

Expressão α	Significado $[\![\alpha]\!]$	(Glosa)
$\texttt{~}\phi$	$\neg[\![\phi]\!]$	(negação)
$\phi_1\ \texttt{\&}\ \phi_2$	$[\![\phi_1]\!]\wedge[\![\phi_2]\!]$	(conjunção)
$\phi_1\ \texttt{\|}\ \phi_2$	$[\![\phi_1]\!]\vee[\![\phi_2]\!]$	(disjunção)
$\phi_1\ \texttt{=>}\ \phi_2$	$[\![\phi_1]\!]\rightarrow[\![\phi_2]\!]$	(implicação)
$\phi_1\ \texttt{<=>}\ \phi_2$	$[\![\phi_1]\!]\leftrightarrow[\![\phi_2]\!]$	(equivalência)
$\texttt{exists}\ x:\phi$	$\exists x([\![\phi]\!])$	(quantificação existencial)
$\texttt{forall}\ x:\phi$	$\forall x([\![\phi]\!])$	(quantificação universal)
$\texttt{existsone}\ x:\phi$	$\exists! x([\![\phi]\!])\equiv\exists x([\![\phi]\!]\wedge\neg\exists y([\![\phi[x/y]]\!]\wedge y\neq x))$	(quantif. existencial única)
$\texttt{edge}(u,v,l,d)$	$u\overset{l}{\longrightarrow}_d v$	(aresta rotulada)
$\texttt{edge}(u,v,d)$	$u\longrightarrow_d v$	(aresta)
$\texttt{dom}(u,v,l,d)$	$u\overset{l}{\rightarrow}\trianglelefteq_d v$	(dominância estrita rotulada)
$\texttt{dom}(u,v,d)$	$u\lhd_d v$	(dominância estrita)
$\texttt{domeq}(u,v,d)$	$u\trianglelefteq_d v$	(dominância relaxada)
$u\ \texttt{<}\ v$	$u\prec v$	(precedência linear)
$v.d.\texttt{entry}.a$	$va(v)(d)(a)$	(atributo lexical)
$v.d.\texttt{attrs}.a$	$va(v)(d)(a)$	(atributo ordinário)
$v.\texttt{word}=s$	$vw(v)=s$	(verificação de palavra)
$\varepsilon_1\ \texttt{=}\ \varepsilon_2$	$[\![\varepsilon_1]\!]=[\![\varepsilon_2]\!]$	(igualdade)
$\varepsilon_1\ \texttt{~=}\ \varepsilon_2$	$[\![\varepsilon_1]\!]\neq[\![\varepsilon_2]\!]$	(desigualdade)
$\varepsilon_1\ \texttt{in}\ \varepsilon_2$	$[\![\varepsilon_1]\!]\in[\![\varepsilon_2]\!]$	(pertinência)
$\varepsilon_1\ \texttt{notin}\ \varepsilon_2$	$[\![\varepsilon_1]\!]\notin[\![\varepsilon_2]\!]$	(não-pertinência)
$\varepsilon_1\ \texttt{subseteq}\ \varepsilon_2$	$[\![\varepsilon_1]\!]\subseteq[\![\varepsilon_2]\!]$	(inclusão)

ε_1 **disjoint** ε_2	$[\![\varepsilon_1]\!] \cap [\![\varepsilon_2]\!] = \varnothing$	(disjunção de conjuntos)
ε :: τ	$[\![\varepsilon]\!]$	(anotação de tipo[13])

```
defprinciple "principle.agreementPW" {
   dims {D}
   constraints {
      forall V: forall V1: forall L:
         edge(V V1 L D) & L in V.D.entry.agree =>
            V.D.attrs.agr = V1.D.attrs.agr
   }
}
```

Figura 22: versão em PL do princípio `agreement`, extraído da distribuição do XDK

IV.4 Considerações finais

Neste capítulo, pretendemos oferecer ao leitor uma visão geral do que é a XDG e do que significa desenvolver uma gramática no sistema XDK. Em especial, esperamos ter demonstrado o apelo da abordagem, no contexto de nossa tese, e quão pouco usual ela é, em geral. Talvez sua maior excentricidade resida no fato de seguir estritamente o paradigma modelo-teórico[14] de análise lingüística (Pullum & Scholz 2001), sem qualquer traço de gerativismo. A XDG tira o foco do como enumerar sentenças bem-formadas, abandonando inclusive a noção de regra de produção, e se concentra na descrição de uma teoria (lógica) unificada das análises/estruturas bem-formadas, de que as sentenças são apenas parte.

Nos capítulos seguintes, trataremos das diferentes contribuições que fizemos ao aplicar a XDG à GLN.

---

[13] Anotações de tipo são geralmente desnecessárias no PW, pois este realiza inferência automática de tipos.

[14] Tradução (bastante anglicista, mas provavelmente a mais eficaz) do termo *model-theoretic* já utilizada em outros trabalhos em português.

V Tornando o *Principle Writer* uma Realidade Prática

Parte significativa das contribuições deste doutorado não se refere diretamente à GLN, mas consiste em melhorias ao XDK que, apesar de terem caráter genérico, tornam-no mais próximo do que consideramos ser uma plataforma ideal de GLN. Muito desse esforço se concentrou num objetivo único, qual seja tornar o *Principle Writer* uma realidade prática. Nossa experimentação com a XDG na GLN precede em muito o lançamento do PW e, sem dúvida, sofreu com isso. Trabalhar com o XDK numa aplicação para a qual não tinha sido ainda utilizado exigiu o desenvolvimento de novos princípios, o que, anteriormente ao PW, era algo difícil, pouco produtivo e sujeito a erros. Como veremos em maior detalhe ao longo deste capítulo, a diferença entre a linguagem então utilizada e a PL, descrita sucintamente na Tabela 2 (pág. 80), vai bem além da mera sintaxe ou do fato de então estarmos manipulando uma API[15] num sistema de programação de propósito geral. Tratava-se de uma linguagem razoavelmente mais complexa, em cujas fórmulas o significado lógico ficava pouco aparente, e que refletia muito — leia-se "dependia muito de" — como o XDK foi implementado. Por outro lado, a PL representa a abordagem oposta, sendo bastante compacta, especializada, abstrata e explicitamente lógica.

Para fins de ilustração, a Figura 23 apresenta a parte principal de uma definição nativa[16] do princípio **`agreement`**. Pode-se pressentir o salto em produtividade/instanciabilidade possibilitado pelo PW quando se compara esse código com o fonte PL equivalente que se encontra na Figura 22 (pág. 81) e se considera (i) que, para definir nativamente esse princípio, ainda é necessário fornecer um arquivo auxiliar de definição de interface, que também é gerado au-

---

[15] *Application Programming Interface*, ou uma interface oferecida por um módulo de *software* para a programação de aplicações que o utilizem.

[16] Isto é, em Oz, a linguagem em que o XDK está implementado.

tomaticamente pelo PW, e (ii) que esse é um dos poucos exemplos de definição nativa que é uma tradução particularmente direta e clara de uma definição em PL.

```
proc {Constraint Nodes G Principle FD FS Select}
  DVA2DIDA = Principle.dVA2DIDA
  ArgRecProc = Principle.argRecProc
  DIDA = {DVA2DIDA 'D'}
  DIDA2LabelLat = G.dIDA2LabelLat
  LabelLat = {DIDA2LabelLat DIDA}
  LAs = LabelLat.constants
in
  if {Helpers.checkModel 'Agreement.oz' Nodes [DIDA#daughtersL]}
  then
    for V in Nodes do
      for V1 in Nodes do
        for L1A in LAs do
          if {Opti.isIn V1.index V.DIDA.model.daughtersL.LA}
             =='out' then skip else
            L1I = {LabelLat.aI2I L1A}
            AgreeLM = V.DIDA.entry.agree
            AgrD = V.DIDA.attrs.agr
            Agr1D = V1.DIDA.attrs.agr
          in
            {FD.impl
             {FS.reified.include V1.index
              V.DIDA.model.daughtersL.L1A}
             {FD.impl
              {FS.reified.include L1I AgreeLM}
              {FD.reified.equal AgrD Agr1D}} 1}
          end
        end
      end
    end
  end
end
```

Figura 23: cerne de uma definição nativa do princípio **`agreement`**, equivalente ao fonte PL da Figura 22 (pág. 81)

Como prova de conceito, Debusmann desenvolveu a gramática **`dissPW.ul`**, uma versão de **`diss.ul`** que usa exclusivamente princípios escritos no PW e é também distribuída com o XDK. Com isso, ele demonstrou não só que a PL é suficientemente expressiva, mas também que a eficiência do código gerado pela primeira versão do PW era ordens de grandeza menor que a dos princípios originais, utilizados em **`diss.ul`**. Apesar de a propagação ser equivalente ou, por vezes, até um pouco mais forte, o código gerado automaticamente tinha um custo quase proibitivo, devido ao número substancialmente maior de propagadores que criava. Tal

perspectiva nos era especialmente desanimadora, já que pretendíamos usar intensivamente o PW em gramáticas com um número razoavelmente maior de dimensões que `dissPW.ul` e em problemas envolvendo muito mais vértices que os presentes nos casos de teste dessa gramática.

Comparando-se a saída do PW com código equivalente escrito à mão, ficava claro que o sistema realizava um mapeamento muito direto entre a PL e a linguagem-alvo (Oz), gerando código muito pouco eficiente. Trata-se de uma situação clássica enfrentada na compilação de linguagens de programação de alto nível, em que um modelo geral de tradução deve ser abandonado sempre que o compilador consegue detectar certas propriedades num trecho do código-fonte que permitam aplicar um modelo alternativo, menos genérico, mas que gere código-objeto mais eficiente. Tais traduções especializadas, otimizadas para uma dada plataforma de execução, são ditas **otimizações**; e, ao aplicá-las, o compilador assume o papel de **otimizador de código**. Talvez o exemplo mais comum e abrangente de otimização seja a de recursão de cauda realizada por qualquer compilador de linguagens lógicas e funcionais, que mapeia certos trechos recursivos do código-fonte em trechos iterativos de código-objeto, evitando uma complexidade de espaço que predisponha a estouros de pilha.

Este capítulo trata exatamente de como conseguimos sanar essa questão relativa à complexidade do código gerado pelo PW, tornando esse módulo do XDK aplicável na prática e, por conseguinte, aumentando a instanciabilidade do sistema como um todo. Para isso, desenvolvemos o ***QuadOptimizer* (QO)**, um otimizador que converte PL em código comparável ao que seria gerado manualmente por um programador experiente. Hoje o QO integra a distribuição do XDK[17] e é a principal razão por que o doutorando esteja citado como autor desse sistema. Abordaremos a descrição do QO da seguinte forma: primeiramente, trataremos, de

---

[17] A partir da versão 1.7.0.

forma abstrata, das otimizações mais relevantes implementadas; em seguida, apresentaremos uma avaliação experimental do desempenho do código otimizado.

V.1 Otimizações

Para explicar a ação do QO, cumpre apresentar como este se insere no PW e quais são os tipos de sua entrada e saída. A Figura 24 responde esquematicamente a essas questões num DFD de nível 0 do PW. Primeiramente, o código-fonte PL é analisado sintaticamente, gerando-se uma representação arbórea semanticamente idêntica à entrada, porém sem ambigüidades e mais apropriada ao processamento subseqüente, dita **PL Arbórea (PLA)**. A estrutura PLA é passada ao **Tipador**, um módulo que anota todas as subexpressões com os seus respectivos tipos, caso haja dados suficientes para inferi-los e não exista nenhum erro de tipagem; caso contrário, mensagens de erro apropriadas são geradas. Vale notar que tais anotações já são exprimíveis mesmo em PL; entretanto, o PW as deixa como optativas para permitir um código mais conciso e fluido, contanto que as haja suficientes para permitir que o Tipador complete sua tarefa. Uma PLA enriquecida com os resultados da inferência de tipos é, portanto, a entrada do QO, que então realiza transformações em porções da estrutura de entrada que consiga identificar como passíveis de otimização. Assim, a saída do QO pode bem ser tanto idêntica à entrada, quando não houver otimizações aplicáveis, como uma versão parcialmente modificada da mesma. O tipo da saída é um superconjunto da PLA, dito **PLA Estendida (PLAE)**, pois contempla todos os constructos desta e ainda outros exclusivos das otimizações, isto é, que não são diretamente exprimíveis em PLA nem na própria PL, portanto. Por fim, a estrutura PLAE é passada para o módulo **Avaliador**, encarregado da geração de código-objeto.

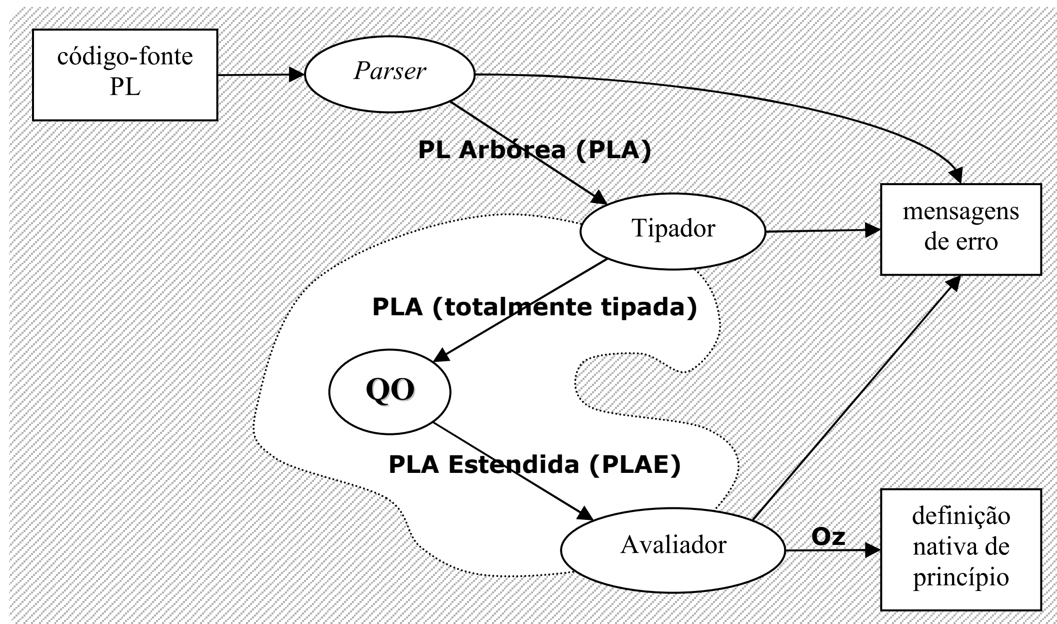

Figura 24: DFD de nível 0 do PW destacando o ponto de inserção do QO

Para a presente discussão, o que mais importa é atentar que o QO recebe uma entrada que, para todos os efeitos, é equivalente à PL de entrada e gera uma saída que pode ser representada numa versão ampliada da PL. Isso nos permite, para efeito da descrição de nossas otimizações, deixar de lado a PLA e a PLAE e recorrer apenas à PL e uma PLE hipotética, ambas mais fáceis ao olho humano.

Descreveremos a seguir, em seções distintas, as duas grandes classes de otimização cobertas pelo QO, a saber: a eliminação de quantificadores e a eliminação de subexpressões comuns.

V.1.1 Eliminação de quantificadores

Uma fonte de complexidade

Em decorrência dos detalhes de implementação do XDK, toda expressão de quantificação PL — isto é, qualquer fórmula do tipo **`exists`** $x{:}\phi$, **`forall`** $x{:}\phi$ e **`existsone`** $x{:}\phi$ — contribui com um fator multiplicativo no cálculo do custo computacional total de um princípio, qual seja a cardinalidade do universo da variável quantificada. Algebricamente, para todo quantificador Q da PL, vale a seguinte propriedade:

$$custo(\phi) \in \Theta(f) \leftrightarrow custo(Q\ x : \phi) \in \Theta(|\mathbb{U}_x|\, f) \quad \text{(V.1)}$$

Grosso modo, isso ocorre porque cada expressão de quantificação é, em última análise, expandida como apresentado nas seguintes equivalências:

$$\begin{aligned} \exists x(\phi) &\equiv \phi[x/x_1] \vee \phi[x/x_2] \vee \ldots \vee \phi[x/x_n] \\ \forall x(\phi) &\equiv \phi[x/x_1] \wedge \phi[x/x_2] \wedge \ldots \wedge \phi[x/x_n] \end{aligned} \quad \text{(V.2)}$$

onde $\{x_1, x_2, \ldots, x_n\} = \mathbb{U}_x$. Ou ainda, de uma forma que melhor espelha a implementação do XDK:

$$\begin{aligned} \exists x(\phi) &\equiv |setify(\exists x(\phi))| \geq 1 \\ \forall x(\phi) &\equiv |setify(\forall x(\phi))| = |\mathbb{U}_x| \\ \exists! x(\phi) &\equiv |setify(\exists! x(\phi))| = 1 \end{aligned} \quad \text{(V.3)}$$

onde $setify(Q\ x : \phi)$ é um conjunto sujeito às seguintes restrições:

$$\begin{aligned} &\varnothing \subseteq setify(Q\ x : \phi) \subseteq \mathbb{U}_x \\ &x_1 \in setify(Q\ x : \phi) \leftrightarrow \phi[x/x_1] \\ &x_2 \in setify(Q\ x : \phi) \leftrightarrow \phi[x/x_2] \\ &\ldots \\ &x_n \in setify(Q\ x : \phi) \leftrightarrow \phi[x/x_n] \end{aligned} \quad \text{(V.4)}$$

Considerando-se que tais universos $\mathbb{U}_x$ costumem ser o conjunto de vértices da análise, ou seja, que as variáveis em questão não raro representem vértices, temos nos quantificadores uma fonte expressiva de complexidade.

Restrições sobre conjuntos finitos

As restrições em (V.4) e os lados direitos das equivalências em (V.3) têm uma realização bastante direta em Oz, utilizando dois recursos poderosos da linguagem que deveremos discutir brevemente para dar ao leitor subsídios à compreensão de por que certas otimizações do QO

sejam eficazes. O primeiro deles é o sistema de **restrições sobre conjuntos finitos**, que permite que uma variável represente um subconjunto finito de $\mathbb{N}$, como seria o caso de $setify(Q\,x:\phi)$ na implementação de (V.4). As restrições básicas para uma tal variável *S* são de dois tipos, a saber: (i) $dom(|S|) \subseteq D$, ou seja, uma restrição sobre o número de elementos de *S*, e (ii) $dom(S) \subseteq \{X : LowerB \subseteq X \subseteq UpperB\}$ onde *LowerB(ound)* e *UpperB(ound)* são valores conhecidos, ditos **limites inferior e superior** de *S*, respectivamente. Por motivo de legibilidade, representaremos as restrições básicas sobre conjuntos da seguinte forma: $|S| \in D$, $LowerB \subseteq S \subseteq UpperB$ e ainda $LowerB \subseteq S$ ou $S \subseteq UpperB$, quando *UpperB* ou *LowerB* não for relevante.

Naturalmente, não só deve haver uma variedade de predicados para relacionar duas ou mais variáveis-conjunto, como também os sistemas de restrições sobre inteiros e conjuntos finitos devem estar interligados. Geralmente, essa interface é realizada por um repertório de primitivas, as mais simples das quais denotam $I \in S$ e $I \notin S$. Note que, agora, estamos relacionando duas variáveis possivelmente não-determinadas e há propagação de restrições. Por exemplo, o propagador de $I \in S$, ao saber de uma nova restrição básica $S \subseteq UpperB$, certamente publicará $dom(I) \subseteq UpperB$. Por outro lado, se *I* for determinado como i_0, então o propagador inferirá $LowerB \cup \{i_0\} \subseteq S$. No sistema Mozart/Oz, se a restrição básica $|S| = 1$ vale, então $I \in S$ é reduzido à restrição não-básica $S \subseteq dom(I)$.

Restrições reificadas

O outro recurso poderoso utilizado na implementação de (V.4) é a reificação de restrições. Dada uma restrição qualquer *C*, define-se a **reificação** de *C* como uma variável $reify(C)$ sujeita às seguintes restrições:

$$
\begin{aligned}
&reify(C) \in \{0,1\} \\
&(reify(C) = 1) \leftrightarrow C
\end{aligned}
\qquad \text{(V.5)}
$$

Em outras palavras, a variável $reify(C)$ coincide com o valor-verdade de *C*. Mais uma vez, a noção de propagação de restrições se aplica. Caso as restrições básicas conhecidas neguem/subsumam *C*, então $reify(C)$ é determinada; se, por outro lado, o valor de $reify(C)$ fica conhecido primeiro, então é imposta ou a restrição *C*, para $reify(C) = 1$, ou $\neg C$, caso contrário.

Um sistema de PCR pode oferecer diretamente um mecanismo genérico de reificação – o próprio operador *reify* – aplicável a qualquer restrição por complexa que seja. Tais mecanismos, entretanto, costumam comprometer a propagação, porque, conquanto seja relativamente simples negar a semântica declarativa de uma restrição, o mesmo não vale para sua semântica operacional, que pode ser composta das semânticas operacionais de diversos propagadores. Em outras palavras, quando todas as variáveis envolvidas estiverem determinadas, o algoritmo subjacente à restrição complexa necessariamente acusará inconsistência ou não, um resultado que pode, então, ser negado. Em contrapartida, o sistema de resolução não é usualmente capaz de reverter esse algoritmo (definido pelo programador) de modo que passe a fazer inferências para a negação de sua restrição original[18]. Por esse motivo, sistemas como o Mozart/Oz oferecem, além de um mecanismo genérico – mas fraco – de reificação, **versões pré-reificadas** de diversos predicados impositores de restrições.

São exatamente tais versões que seriam utilizadas na tradução de (V.4), tanto para o operador de pertinência $\in$ quanto para as potencialmente diversas subexpressões constituintes da fórmula ϕ. Considere (V.6) abaixo, uma restrição nos moldes das impostas por (V.4), contendo, contudo, uma restrição complexa como operando direito da equivalência. Assumindo-se que

---

[18] Uma forma de usar um operador de reificação genérico e obter máxima propagação consiste em o programador, para cada restrição complexa *C* a ser reificada, definir explicitamente a restrição $\neg C$, reificar a ambas por B_1 e B_2 respectivamente e impor $B_1 \leftrightarrow \neg B_2$.

as três restrições ϕ_1, ϕ_2 e ϕ_3 sejam atômicas, (V.6) deveria ser implementada por meio de reificação, como explicitado em (V.7).

$$x_1 \in S \leftrightarrow \phi_1 \wedge (\phi_2 \vee \phi_3) \tag{V.6}$$

$$1 = \left(reify\left(x_1 \in S\right) \widehat{\leftrightarrow} reify(\phi_1) \mathbin{\widehat{\wedge}} \left(reify(\phi_2) \mathbin{\widehat{\vee}} reify(\phi_3)\right)\right) \tag{V.7}$$

Na equação (V.7), denotam-se por $\widehat{\leftrightarrow}$, $\widehat{\wedge}$ e $\widehat{\vee}$ versões reificadas dos operadores lógicos em questão que recebem por operandos variáveis inteiras modelando valores-verdade. Formalmente, para todo operador lógico $\odot$:

$$\begin{array}{ll} p \mathbin{\widehat{\odot}} q \equiv reify\left(p = 1 \odot q = 1\right) & (\odot \text{ binário}) \\ \widehat{\odot}\, p \equiv reify\left(\odot\left(p = 1\right)\right) & (\odot \text{ unário}) \end{array} \tag{V.8}$$

Uma oportunidade

Agora que já dispomos da intuição da forma como os quantificadores da PL sejam traduzidos para Oz e do custo que representam, discutiremos formas alternativas mais eficientes — e, portanto, otimizadas — de implementar algumas expressões de quantificação. De fato, ao comparar definições de princípio nativas escritas por programadores experientes com o código gerado pelo PW, fica patente o uso muito menos freqüente de expansões como as presentes em (V.2) e (V.3). Isso ocorre pelo simples motivo de que o XDK implementa cada projeção da análise em construção como um grafo independente e cada um desses grafos como uma coleção de variáveis-conjunto, ditas **gráficas**. Em específico, para cada dimensão gráfica *d*, cada vértice *v* tem definido para si um *kit* de variáveis gráficas como descrito na Tabela 3.

Tabela 3: variáveis-conjunto de vértices, ditas **gráficas**, que definem as projeções de uma análise XDG segundo o XDK

Variável	Definição	Glosa
$eq(v,d)$	$\{v\}$	identidade

$mothers(v,d)$	$\{u : u \rightarrow_d v\}$	pais
$daughters(v,d)$	$\{u : v \rightarrow_d u\}$	filhos
$mothersL(v,l,d)$	$\left\{u : u \overset{l}{\rightarrow}_d v\right\}$	pais que atingem *v* por arestas de rótulo *l*
$daughtersL(v,l,d)$	$\left\{u : v \overset{l}{\rightarrow}_d u\right\}$	filhos atingidos por arestas de rótulo *l*
$up(v,d)$	$\{u : u \lhd_d v\}$	ancestrais
$down(v,d)$	$\{u : v \lhd_d u\}$	descendentes
$equp(v,d)$	$\{u : u \unlhd_d v\}$	$up(v,d) \cup \{v\}$
$eqdown(v,d)$	$\{u : v \unlhd_d u\}$	$down(v,d) \cup \{v\}$
$upEndL(v,d)$	$\left\{u : u \overset{l}{\rightarrow}\unlhd_d v\right\}$	ancestrais que atingem *v* por caminho iniciado com rótulo *l*
$downL(v,l,d)$	$\left\{u : v \overset{l}{\rightarrow}\unlhd_d u\right\}$	descendentes atingidos por caminho iniciado com rótulo *l*
$labels(v,d)$	$\left\{l : \exists u\left(u \overset{l}{\rightarrow}_d v\right)\right\}$	rótulos das arestas de entrada

Trata-se de um conjunto de variáveis definidas e inter-relacionadas em decorrência de manifestos de grafo como o presente na Figura 11 (pág. 65). Porquanto elas estão disponíveis sempre que um princípio trate de propriedades gráficas, os programadores de princípios nativos preferem manipulá-las diretamente, gerando fórmulas muito mais eficientes e concisas que as possibilitadas pelas expansões de quantificadores vistas. Por exemplo, uma fórmula PL como:

```
forall V: forall U: forall L:
  edge(V U L D1) => edge(V U L D2)
```
(V.9)

— que basicamente estabelece que, para todo grafo *g*, o conjunto de arestas em $\prod_{d_1} g$ deve ser um subconjunto das arestas em $\prod_{d_2} g$ — pode ser implementada de forma extremamente eficiente por código equivalente a:

$$\forall v \forall l \left(daughtersL(v,l,d_1) \subseteq daughtersL(v,l,d_2) \right) \qquad \text{(V.10)}$$

ou, em PLE:

```
forall V: forall L:
  daughtersL(V L D1) subseteq daughtersL(V L D2)
```
(V.11)

A transformação de (V.9) em (V.11) é dita uma **Eliminação de Quantificadores (EQ)** e é um bom exemplo do que pretendemos realizar com o QO. Trata-se da geração de uma expressão PLAE ϕ' semanticamente equivalente a uma dada ϕ tal que (i) o número de quantificadores de ϕ' seja menor que o de ϕ (o caso ideal) ou (ii) que ao menos uma subexpressão φ de ϕ estritamente dependente de um conjunto $\{v_1, v_2, ..., v_n\}$ de variáveis tenha sido substituída por outra φ' em que uma ou mais $\{v_i\}$ não sejam mais mencionadas e que possa ser avaliada fora do escopo dos quantificadores das $\{v_i\}$. Neste último caso, pode-se dizer que terá havido uma **EQ parcial**, pois se evitará a multiplicação desnecessária do custo de φ, pelo menos. Em ambos os casos, vale observar a EQ pode ser entendida como uma **eliminação de variáveis**.

A EQ é, de longe, a tarefa mais complexa que o QO realiza, o que é compensado pelo fato de ser também a que mais impacto tem sobre custo computacional, especialmente se as variáveis eliminadas representarem vértices. Cumpre agora identificar classes as mais abrangentes possíveis de fórmulas PLE que sejam passíveis de EQ e uma forma de construir as formas otimizadas correspondentes.

EQ básica

A classe mais simples, mais freqüente e, portanto, mais importante de fórmulas passíveis de EQ são aquelas que permitem aplicar uma das seguintes equivalências simples:

$$\begin{aligned}
&\exists x(\phi) \equiv \left|\{x : \phi\}\right| \geq 1 \\
&\forall x(x \in \mathbb{U}_x \rightarrow \phi) \equiv \left|\{x : x \in \mathbb{U}_x \wedge \phi\}\right| = \left|\mathbb{U}_x\right| \qquad \text{(V.12)} \\
&\exists ! x(\phi) \equiv \left|\{x : \phi\}\right| = 1
\end{aligned}$$

Não por acaso, tais equivalências guardam semelhança notável com as presentes em (V.3); contudo, a grande diferença reside na construção das variáveis-conjunto $\{x : ...\}$ acima, que não farão uso da operação *setify*, *mas serão obtidas como uma combinação direta de variáveis gráficas* por meio de operações/restrições sobre conjuntos. A fórmula (V.9) cai dentro dessa classe, embora (V.11) talvez não pareça poder ser obtida por meio de (V.12). Entretanto, caso consideremos a equivalência:

$$\begin{gathered}
A \subseteq B \\
\equiv \\
\left|\overline{A} \cup B\right| = \left|\mathbb{U}\right|
\end{gathered} \qquad \text{(V.13)}$$

onde $\mathbb{U}$ é o conjunto universo e $\overline{X} = \mathbb{U} - X$, é trivial realizarmos a derivação de (V.9) em (V.11), como segue. Por serem mais concisas, operaremos com as representações lógicas das expressões PLE envolvidas, partindo de (V.14), equivalente a (V.9) e em que o operador de implicação está grafado como "$\Rightarrow$" para evitar confusão com o operador de aresta. Primeiramente, reordenamos os quantificadores, obtendo (V.15) abaixo.

$$\forall v \forall u \forall l \left(u \in V \wedge \left(v \overset{l}{\longrightarrow}_{d_1} u \Rightarrow v \overset{l}{\longrightarrow}_{d_2} u \right) \right) \qquad \text{(V.14)}$$

$$\forall v \forall l \left(\forall u \left(u \in V \wedge \left(v \overset{l}{\longrightarrow}_{d_1} u \Rightarrow v \overset{l}{\longrightarrow}_{d_2} u \right) \right) \right) \qquad \text{(V.15)}$$

Na seqüência, ao aplicar (V.12) a (V.15), obtemos o seguinte:

$$\forall v \forall l \left(\left| \left\{ u : u \in V \wedge \left(v \overset{l}{\longrightarrow}_{d_1} u \Rightarrow v \overset{l}{\longrightarrow}_{d_2} u \right) \right\} \right| = \left|\mathbb{V}\right| \right) \qquad \text{(V.16)}$$

onde $\mathbb{V}$, o conjunto de vértices do grafo, faz as vezes de universo. Aplicando a equivalência básica (V.17), obtemos (V.18).

$$\neg\phi_1 \vee \phi_2 \equiv \phi_1 \rightarrow \phi_2 \qquad \text{(V.17)}$$

$$\forall v \forall l \left(\left| \left\{ u : u \in V \wedge \left(\neg v \overset{l}{\rightarrow}_{d_1} u \vee v \overset{l}{\rightarrow}_{d_2} u \right) \right\} \right| = |\mathbb{V}| \right) \qquad \text{(V.18)}$$

Em seguida, convertemos o termo $\{u : u \in \mathbb{V} \wedge ...\}$ de (V.18) numa fórmula que envolva apenas variáveis gráficas, como visto em (V.19).

$$\forall v \forall l \left(\left| \overline{daughtersL(v,l,d_1)} \cup daughtersL(v,l,d_2) \right| = |\mathbb{V}| \right) \qquad \text{(V.19)}$$

Por fim, aplicamos (V.13) a (V.19) e obtemos (V.10), que, por sua vez, é equivalente a (V.11).

O passo crucial de toda EQ — básica ou não — está representado na passagem de (V.18) para (V.19), isto é, a tradução de uma fórmula lógica ϕ numa expressão de conjunto α tal que (i) uma variável qualquer *u* presente em ϕ não seja mencionada em α e que (ii) α cubra todos e somente os valores de *u* legitimados por ϕ. Denotaremos tal tradução por $\{\phi\}_u = \alpha$, dita a **eliminação** da variável *u* da fórmula ϕ, ou simplesmente uma **eliminação de variável**.

Tabela 4: regras básicas de cálculo de $\{\phi\}_x$, a eliminação da variável *x* da fórmula ϕ

Categoria	Constructo PL Original	Regra de Cálculo	Constructo PLE Gerado
operadores lógicos	`~`	$\{\neg\phi\}_x = \overline{\{\phi\}_x}$	`compl`
	`&`	$\{\phi_1 \wedge \phi_2\}_x = \{\phi_1\}_x \cap \{\phi_2\}_x$	`intersect`
	`|`	$\{\phi_1 \vee \phi_2\}_x = \{\phi_1\}_x \cup \{\phi_2\}_x$	`union`
	`=>`	$\{\phi_1 \rightarrow \phi_2\}_x = \{\neg\phi_1 \vee \phi_2\}_x = \overline{\{\phi_1\}_x} \cup \{\phi_2\}_x$	NA
	`<=>`	$\{\phi_1 \leftrightarrow \phi_2\}_x = \{(\phi_1 \wedge \phi_2) \vee \neg(\phi_1 \vee \phi_2)\}_x = \left(\{\phi_1\}_x \cap \{\phi_2\}_x\right) \cup \overline{\{\phi_1\}_x \cup \{\phi_2\}_x}$	NA
pertinência	`in`	$\{x \in \alpha\}_x = \alpha$	NA
aresta	`edge`	$\left\{u \overset{l}{\rightarrow}_d v\right\}_x = \begin{cases} mothersL(v,l,d) & (x=u) \\ daughtersL(u,l,d) & (x=v) \end{cases}$	`mothersL` `daughtersL`
		$\{u \rightarrow_d v\}_x = \begin{cases} mothers(v,d) & (x=u) \\ daughters(u,d) & (x=v) \end{cases}$	`mothers` `daughters`
dominância	`dom`	$\left\{u \overset{l}{\rightarrow}\!\trianglelefteq_d v\right\}_x = \begin{cases} upEndL(v,l,d) & (x=u) \\ downL(u,l,d) & (x=v) \end{cases}$	`upEndL` `downL`
		$\{u \rightarrow\!\trianglelefteq_d v\}_x = \begin{cases} up(v,d) & (x=u) \\ down(u,d) & (x=v) \end{cases}$	`up` `down`
	`domeq`	$\{u \trianglelefteq_d v\}_x = \begin{cases} equp(v,l,d) & (x=u) \\ eqdown(u,l,d) & (x=v) \end{cases}$	`equp` `eqdown`
igualdade	`=`	$\{x = \varepsilon\}_x = \{\varepsilon\}$	`asSet`
	`~=`	$\{x \neq \varepsilon\}_x = \{\neg(x = \varepsilon)\}_x = \overline{\{x = \varepsilon\}_x} = \overline{\{\varepsilon\}}$	NA

A

Tabela 4 consiste num conjunto fundamental de regras de cálculo de $\{\phi\}_x$, entre as quais merecem destaque as relativas a propriedades de grafos, quais sejam **`edge`**, **`dom`** e **`domeq`**. Note que, nesses casos, a expressão-fonte pode ser traduzida de duas formas diferentes, dependendo de qual variável esteja sendo eliminada.

Podemos finalmente caracterizar a classe de EQs ora focalizada. É dita **básica** toda EQ obtenível por meio das seguintes equivalências:

$$\begin{array}{ll} \exists x(\phi) \equiv \left|\{\phi\}_x\right| \geq 1 & \equiv oneOrMore\left(\{\phi\}_x\right) \\ \forall x(x \in \mathbb{U}_x \rightarrow \phi) \equiv \left|\{x \in \mathbb{U}_x \wedge \phi\}_x\right| = \left|\mathbb{U}_x\right| & \equiv all\left(\{\phi\}_x, \mathbb{U}_x\right) \\ \exists! x(\phi) \equiv \left|\{\phi\}_x\right| = 1 & \equiv one\left(\{\phi\}_x\right) \end{array} \qquad \text{(V.20)}$$

EQ básica: aprofundamento de quantificadores

Embora (V.20) capture a essência das EQs básicas, uma implementação deve realizar algumas manobras auxiliares para tirar o máximo proveito dessa classe. A primeira delas é a que chamamos de **aprofundamento** do quantificador a ser eliminado e está exemplificada na passagem de (V.14) para (V.15), em uma de suas formas mais simples. Essa manobra é benéfica por uma série de motivos. Em primeiro lugar, o aprofundamento pode evitar fórmulas ϕ para as quais $\{\phi\}_x$ ou não esteja definida ou tenha um custo dispensável. Além disso, observa-se na prática que quanto mais localizada for a transformação operada, maior será a propagação obtida e a probabilidade de que outros quantificadores, originalmente internos ao quantificador eliminado, também sejam passíveis de novas EQs, básicas ou não.

A Tabela 5, Tabela 6 e Tabela 7 compreendem todas as regras seguidas pelo QO no aprofundamento de quantificadores. Cada linha dessas tabelas contém uma equivalência lógica que efetua um passo de aprofundamento e possíveis precondições à sua aplicação. A Tabela 5 reúne o que há de comum a todos os quantificadores. A primeira de suas regras foi a que se aplicou na passagem de (V.14) para (V.15). A segunda e última regra dessa tabela contém

uma precondição importante, que permite inverter a ordem de dois quantificadores diferentes, qual seja que as duas variáveis em questão não ocorram como co-argumentos de nenhum predicado na fórmula aninhada. A Tabela 6, por sua vez, apresenta as regras aplicáveis somente aos quantificadores **`forall`** ($\forall$) e **`exists`** ($\exists$), enquanto a Tabela 7 trata de **`existsone`** ($\exists!$) e o seu dual, **`allbutone`** ($\forall!$). Foi necessário definir este último quantificador para tornar a negação — explícita ou não — "permeável" a **`existsone`**. Uma expressão $\forall! x(\phi)$ é verdadeira se e somente se ϕ é verdadeira para todos os valores de x exceto para um determinado x_0, como formalizado em (V.21) abaixo. A eliminação de **`allbutone`** é efetuada de acordo com (V.22).

$$\forall! x(\phi) \equiv \exists x\left(\neg\phi \wedge \forall y\left(\neg\phi[x/y] \rightarrow y = x\right)\right) \quad \text{(V.21)}$$

$$\forall! x(x \in \mathbb{U}_x \rightarrow \phi) \equiv \left|\overline{\{x \in \mathbb{U}_x \wedge \phi\}_x}\right| = 1 \quad \equiv \quad one\left(\overline{\{x \in \mathbb{U}_x \wedge \phi\}_x}\right) \quad \text{(V.22)}$$

Tabela 5: regras gerais de aprofundamento de quantificadores.

Equivalência	Precondições
$Qx\left(Qy(\phi)\right) \equiv Qy\left(Qx(\phi)\right)$	
$Qx\left(Q'y(\phi)\right) \equiv Q'y\left(Qx(\phi)\right)$	$Q \neq Q'$ x e y não se relacionam em ϕ

Tabela 6: regras de aprofundamento específicas para quantificadores $Q \in \{\forall, \exists\}$.

Equivalência	Obs.	Precondições
$Qx(\neg\phi) \equiv \neg\bar{Q}x(\phi)$	1	
$Qx(\alpha \wedge \phi) \equiv \alpha \wedge Qx(\phi)$	2	x não ocorre em α
$\forall x(\phi_1 \wedge \phi_2) \equiv \forall x(\phi_1) \wedge \forall x(\phi_2)$		
$Qx(\alpha \vee \phi) \equiv \alpha \vee Qx(\phi)$	2	x não ocorre em α
$\exists x(\phi_1 \vee \phi_2) \equiv \exists x(\phi_1) \vee \exists x(\phi_2)$		

$Qx(\alpha \to \phi) \equiv \alpha \to Qx(\phi)$		x não ocorre em α
$Qx(\phi \to \alpha) \equiv \overline{Q}x(\phi) \to \alpha$	1	x não ocorre em α
$\exists x(\phi_1 \to \phi_2) \equiv \forall x(\phi_1) \to \exists x(\phi_2)$		
$\forall x(\alpha \leftrightarrow \phi) \equiv (\alpha \to \forall x(\phi)) \wedge (\exists x(\phi) \to \alpha)$		x não ocorre em α
$\exists x(\alpha \leftrightarrow \phi) \equiv (\alpha \wedge \exists x(\phi)) \vee \neg(\alpha \vee \forall x(\phi))$		x não ocorre em α

Observações:

1. $\overline{\forall} = \exists$, $\overline{\exists} = \forall$;
2. versão comutada, como $Qx(\phi \wedge \alpha) \equiv Qx(\phi) \wedge \alpha$, também aplicada.

Tabela 7: regras de aprofundamento específicas para os quantificadores $Q \in \{\forall!, \exists!\}$.

Equivalência	**Obs.**	**Precondições**
$Qx(\neg\phi) \equiv \overline{Q}x(\phi)$	1	
$Qx(\alpha \wedge \phi) \equiv \alpha \wedge Qx(\phi)$	2	x não ocorre em α
$Qx(\alpha \vee \phi) \equiv \neg\alpha \vee Qx(\phi)$	2	x não ocorre em α
$Qx(\alpha \to \phi) \equiv \alpha \wedge Qx(\phi)$		x não ocorre em α
$Qx(\phi \to \alpha) \equiv \neg\alpha \wedge \overline{Q}x(\phi)$	1	x não ocorre em α

Observações:

1. $\overline{\forall!} = \exists!$, $\overline{\exists!} = \forall!$;
2. versão comutada, como $Qx(\phi \wedge \alpha) \equiv Qx(\phi) \wedge \alpha$, também aplicada.

EQ básica: quantificadores aninhados

Mesmo todas as regras de aprofundamento de quantificadores apresentadas não dariam conta de (V.23) abaixo, que é idêntica a (V.9) salvo pela troca do quantificador da variável **U**.

```
forall V: exists U: forall L:
  edge(V U L D1) => edge(V U L D2)
```
(V.23)

Nesse caso, as regras básicas de cálculo de $\{\phi\}_x$ da

Tabela 4 não são suficientes para permitir a eliminação de variáveis, já que não incluem expressões envolvendo quantificadores. Felizmente, é possível definir novas regras que cubram tais casos, como apresentado na Tabela 8. Não se inclui uma regra para **`allbutone`**, pois esse quantificador nunca é encontrado no código a ser processado, só existindo virtualmente no aprofundamento de um quantificador para ser eliminado em seguida. As regras para **`forall`** e **`exists`** são relativamente simples e consistem em expansões análogas às de (V.2), exceto pelo uso de operadores de conjuntos. Bem mais complexo é o tratamento de **`existsone`**, que envolve encontrar o conjunto de valores de x que tornem ϕ verdadeira para cada $y' \in \mathbb{U}_y$ e nenhum outro $y'' \in \mathbb{U}_y - \{y'\}$. Como **`existsone`** é muito raramente empregado na prática, nossos experimentos não puderam atestar se há vantagem no uso dessa regra. Atualmente o QO permite desabilitar a geração de **`setExistsOne`**.

Tabela 8: regras de cálculo de $\{\phi\}_x$ para expressões envolvendo quantificadores

Categoria	Constructo PL Original	Regra de Cálculo	Constructo PLE Gerado
quantificadores	**`forall`**	$\{\forall y(\phi)\}_x = \{\phi[y/y_1]\}_x \cap \ldots \cap \{\phi[y/y_n]\}_x = \bigcap_{y \in \mathbb{U}_y} \{\phi\}_x$	**`setForall`**
	`exists`	$\{\exists y(\phi)\}_x = \{\phi[y/y_1]\}_x \cup \ldots \cup \{\phi[y/y_n]\}_x = \bigcup_{y \in \mathbb{U}_y} \{\phi\}_x$	**`setExists`**
	`existsone`	$\{\exists! y(\phi)\}_x = \bigcup_{y' \in \mathbb{U}_y} \left(\{\phi[y/y']\}_x - \bigcup_{y'' \in \mathbb{U}(y')} \{\phi[y/y'']\}_x \right)$ onde $\mathbb{U}(y') = \mathbb{U}_y - \{y'\}$	**`setExistsone`**

Por meio das novas regras, o quantificador de **`U`** pode ser eliminado de (V.23), obtendo-se o seguinte:

```
forall V:
   oneOrMore(setForall L:
      compl(daughtersL(V L D1))
      union daughtersL(V L D2))
```
(V.24)

EQ básica: fórmulas livres

Todo o instrumental apresentado até o momento não permite eliminar nenhum quantificador nas expressões abaixo:

```
forall V: forall V1: forall V2:
   edge(V1 V D2) & dom(V2 V D1) => domeq(V2 V1 D1)
```
(V.25)

```
forall V: forall V1: forall L:
   edge(V V1 L D1) & L in V.D3.entry.linkMother
   => edge(V1 V D2)
```
(V.26)

Isso ocorre porque, qualquer que seja a variável selecionada para eliminação, sempre haverá uma expressão para a qual $\{\phi\}_x$ não esteja definida. Por exemplo, se selecionarmos **`V2`** em (V.25), $\{\phi\}_x$ não saberá o que fazer com **`edge(V1 V D2)`**, o mesmo problema ocorrendo entre **`V1`** e a expressão **`L in V.D3.entry.linkMother`** em (V.26). Entretanto, ambas as expressões têm uma propriedade muito especial, a saber: são livres da variável a ser eliminada. Quando tentamos encontrar o conjunto dos valores de uma variável x que tornam verdadeira uma fórmula ϕ livre de x, chegamos ao seguinte resultado:

$$\{x : x \in \mathbb{U}_x \wedge \phi\} = \begin{cases} \{x : \bot\} = \varnothing & (\phi \text{ é falsa}) \\ \{x : x \in \mathbb{U}_x\} = \mathbb{U}_x & (\phi \text{ é verdadeira}) \end{cases} \qquad \text{(V.27)}$$

ou seja, o conjunto em questão depende apenas do universo da variável e do valor-verdade de ϕ, que independe de x. Isso nos permite definir $\{\phi\}_x$ para ϕ livre de x, da seguinte forma:

$$\{\phi\}_x = \phi \otimes \mathbb{U}_x \qquad (x \text{ não ocorre em } \phi) \qquad \text{(V.28)}$$

onde $\phi \otimes S$ é uma variável-conjunto sujeita às seguintes restrições:

$$
\begin{aligned}
&(\phi \otimes S) \subseteq S \\
&(\phi \otimes S) = \varnothing \leftrightarrow \neg\phi \vee S = \varnothing \\
&S \subseteq (\phi \otimes S) \leftrightarrow \phi \vee S = \varnothing
\end{aligned}
\qquad (V.29)
$$

Intuitivamente, $\otimes$ define multiplicação/conjunção entre valores-verdade e conjuntos.

Teoricamente, (V.28) já seria suficiente para tratar adequadamente de fórmulas como (V.25) e (V.26). Contudo, dados experimentais indicam que expressões $\otimes$ podem ser bastante prejudiciais à propagação de restrições. O ideal é ou que elas sejam eliminadas ou que todas as fórmulas livres sejam combinadas numa única fórmula (livre, portanto) que multiplique não o universo, mas a própria expressão-conjunto resultante do processamento das fórmulas não-livres. Denominamos a tarefa de transformar uma expressão-conjunto de modo que se aproxime desse ideal como **alçamento de expressões livres**.

Tabela 9: regras de alçamento de expressões livres

Prioridade	Equivalência	Obs.
1	$\overline{\phi \otimes \mathbb{U}_x} \equiv (\neg\phi) \otimes \mathbb{U}_x$	
	$(\phi_1 \otimes \mathbb{U}_x) \cup (\phi_2 \otimes \mathbb{U}_x) \equiv (\phi_1 \vee \phi_2) \otimes \mathbb{U}_x$	
	$(\phi_1 \otimes \mathbb{U}_x) \cup (\phi_2 \oplus \alpha) \equiv (\phi_1 \vee \phi_2) \oplus \alpha$	*
	$(\phi_1 \otimes \mathbb{U}_x) \cap (\phi_2 \otimes \alpha) \equiv (\phi_1 \wedge \phi_2) \otimes \alpha$	*
2	$(\phi \otimes \mathbb{U}_x) \cup \alpha \equiv \phi \oplus \alpha$	*
	$(\phi \otimes \mathbb{U}_x) \cap \alpha \equiv \phi \otimes \alpha$	*
3	$\overline{\phi \otimes \alpha} \equiv (\neg\phi) \oplus \overline{\alpha}$	
	$\overline{\phi \oplus \alpha} \equiv (\neg\phi) \otimes \overline{\alpha}$	
	$(\phi_1 \otimes \alpha) \cap (\phi_2 \otimes \beta) \equiv (\phi_1 \wedge \phi_2) \otimes (\alpha \cap \beta)$	
	$(\phi_1 \oplus \alpha) \cup (\phi_2 \oplus \beta) \equiv (\phi_1 \vee \phi_2) \oplus (\alpha \cup \beta)$	
4	$(\phi \otimes \alpha) \cap \beta \equiv \phi \otimes (\alpha \cap \beta)$	*
	$(\phi \oplus \alpha) \cup \beta \equiv \phi \oplus (\alpha \cup \beta)$	*

* Versão comutada também aplicada.

A Tabela 9 reúne todas as regras de alçamento aplicadas pelo QO. O primeiro ponto a observar sobre essa tabela é o operador $\oplus$, que surge ao longo do alçamento e, intuitivamente, tem valor aditivo/conjuntivo. Formalmente, $\phi \oplus S$ é uma variável-conjunto sujeita às seguintes restrições:

$$\begin{aligned} & S \subseteq (\phi \oplus S) \subseteq \mathbb{U}_x \\ & (\phi \oplus S) = \mathbb{U}_x \leftrightarrow \phi \vee S = \mathbb{U}_x \\ & (\phi \oplus S) = S \leftrightarrow \neg\phi \vee S = \mathbb{U}_x \end{aligned} \qquad \text{(V.30)}$$

Como é comum que diferentes regras se apliquem a um mesmo caso, há uma ordem de prioridade — também indicada na Tabela 9 — que é usada como critério de desempate. As regras de mais alta prioridade (1 e 2) lidam com os casos iniciais, em que o universo ainda é um operando e que têm propriedades combinatórias particularmente interessantes. As regras de prioridade 3 só envolvem operandos com $\otimes$ e $\oplus$ e ainda conseguem separar totalmente fórmulas livres de expressões-conjunto, propiciando, como as regras anteriores, um alçamento ótimo. Por fim, nas regras de prioridade 4, β é possivelmente uma expressão envolvendo $\otimes$ e $\oplus$ que não terá sua fórmula livre alçada, mas ainda se alça ϕ. Naturalmente, existem casos que não se enquadram em nenhumas das regras e não mudam durante o alçamento, como expressões do tipo $(\phi_1 \oplus \alpha) \cap (\phi_2 \oplus \beta)$.

O alçamento é realizado pela aplicação exaustiva das regras da Tabela 9 sobre a árvore retornada por $\{\phi\}_x$, de maneira *bottom-up*, ou seja, transformando um dado nó apenas quando suas subárvores tiverem sido totalmente processadas. Por fim, caso o operador mais externo da expressão resultante seja $\otimes$ ou $\oplus$, aplicam-se as seguintes equivalências:

$$
\begin{aligned}
&\exists x(\phi) \equiv \begin{cases} \phi' \wedge oneOrMore(\alpha) & \left(\{\phi\}_x = \phi' \otimes \alpha\right) \\ \phi' \vee oneOrMore(\alpha) & \left(\{\phi\}_x = \phi' \oplus \alpha\right) \end{cases} \\
&\forall x(x \in \mathbb{U}_x \to \phi) \equiv \begin{cases} \phi' \wedge all\left(\alpha, \mathbb{U}_x\right) & \left(\{x \in \mathbb{U}_x \wedge \phi\}_x = \phi' \otimes \alpha\right) \\ \phi' \vee all\left(\alpha, \mathbb{U}_x\right) & \left(\{x \in \mathbb{U}_x \wedge \phi\}_x = \phi' \oplus \alpha\right) \end{cases} \\
&\exists! x(\phi) \equiv \begin{cases} \phi' \wedge one(\alpha) & \left(\{\phi\}_x = \phi' \otimes \alpha\right) \\ \neg\phi' \wedge one(\alpha) & \left(\{\phi\}_x = \phi' \oplus \alpha\right) \end{cases} \\
&\forall! x(x \in \mathbb{U}_x \to \phi) \equiv \begin{cases} \phi' \wedge one(\overline{\alpha}) & \left(\{x \in \mathbb{U}_x \wedge \phi\}_x = \phi' \otimes \alpha\right) \\ \neg\phi' \wedge one(\overline{\alpha}) & \left(\{x \in \mathbb{U}_x \wedge \phi\}_x = \phi' \oplus \alpha\right) \end{cases}
\end{aligned}
\tag{V.31}
$$

Dessa forma, é eventualmente possível eliminar esses operadores especiais, como acontece quando o QO processa as fórmulas (V.25) e (V.26) da página 101, gerando (V.32) e (V.33), respectivamente.

```
forall V: forall V1:
   edge(V1 V D2)
   => up(V D1) subseteq equp(V1 D1)
```

(V.32)

```
forall V: forall L:
   L in V.D3.entry.linkMother
   => daughters(V L D1) subseteq mothers(V D2)
```

(V.33)

EQ básica: ajustes superficiais

Nas equações (V.32) e (V.33), também se observam os resultados do passo final da EQ básica, que usa as equivalências (V.13), à página 94, e (V.34), abaixo, para gerar respectivamente os operadores de inclusão e igualdade de conjuntos.

$$
\begin{gathered}
A = B \\
\equiv \\
\left|\left(A \cap B\right) \cup \overline{\left(A \cup B\right)}\right| = \left|\mathbb{U}\right|
\end{gathered}
\tag{V.34}
$$

EQ de seleção

Isolamos duas outras classes de fórmulas PLE que são passíveis de EQ, dita agora **de seleção** porque emprega as assim chamadas **restrições de seleção**. O que torna essas classes interessantes é exatamente o tratamento muito eficiente dado por ambientes de PCR — em específico, o Mozart/Oz — a essas restrições, que incluem os seguintes tipos:

$$\boxed{S = \bigcup_{Inds} V} \equiv S = \bigcup_{i \in Inds} V_i \tag{V.35}$$

$$\boxed{A = \bigcap_{Inds} V} \equiv A = \bigcap_{i \in Inds} V_i \tag{V.36}$$

onde V é um vetor de conjuntos $(V_1, V_2, ..., V_n)$ e *Inds* é um conjunto de índices em V. O especial, nesse caso, é que as primitivas do sistema impositoras de (V.35) e (V.36) permitem que A, todos os V_i e *Inds* sejam variáveis-conjunto e realizam propagação de restrições em todos os sentidos entre essas variáveis.

Para a aplicação das otimizações de que trataremos a seguir, o conjunto de vértices $\mathbb{V}$ de toda análise deve ter as seguintes propriedades:

$$\begin{aligned} &\{1\} \subseteq \mathbb{V} \subset \mathbb{N}^* \\ &\forall i, j\left(\{i, j\} \subseteq \mathbb{V} \wedge i < j \rightarrow \forall k\left(k \in \mathbb{N}^* \wedge i < k < j \rightarrow k \in \mathbb{V}\right)\right) \end{aligned} \tag{V.37}$$

ou seja, $\mathbb{V}$ deve conter os números inteiros de 1 a $|\mathbb{V}|$. Não por coincidência, tal é exatamente a forma como o XDK implementa esses conjuntos.

Antes de prosseguirmos, cabe uma palavra de advertência: as EQs de seleção, como descritas abaixo, ainda não foram completamente incorporadas ao QO e não participam dos experimentos descritos na seção de avaliação. Entretanto, incluímos sua formalização neste capítulo porque a consideramos, por si só, um resultado extremamente interessante.

EQ de seleção: expressões de pertinência

A primeira classe de fórmulas de que trataremos está caracterizada por (V.38) abaixo:

$$\exists v\left(\beta \vee \alpha_1\phi_1 \vee \alpha_2\phi_2 \vee \ldots \vee \alpha_n\phi_n\right)$$

$$\boxed{\begin{array}{l} \forall i,\ 1 \le i \le n: \\ \quad \exists\{\alpha_i\}_v \wedge \\ \quad \boxed{\begin{array}{l} \exists m, k, \varepsilon, acc_1 \ldots acc_m: \\ \qquad \phi_i = \bigwedge_{j=1}^{k}\left(\varepsilon \in acc_j(v)\right) \wedge \bigwedge_{j=k}^{m}\left(\neg\left(\varepsilon \in acc_j(v)\right)\right) \end{array}} \end{array}} \qquad \text{(V.38)}$$

onde todo $acc_j(v)$ é uma expressão simples de acesso a um atributo de *v*, o que, em PL, se realiza como $\mathtt{v}.d_j.\mathtt{attr}.a_j$ para alguma dimensão d_j e identificador de atributo a_j. Vale notar que, no tocante à estrutura imposta pelos operadores de conjunção, disjunção e negação, toda fórmula em **Forma Disjuntiva Normal** (**DNF**, do inglês *Disjunctive Normal Form*) satisfaz às condições de (V.38). Como toda fórmula pode ser passada para a DNF[19], (V.38) quer dizer que predicados do tipo $\varepsilon \in acc_j(v)$ não mais representam necessariamente um empecilho à EQ. A maior restrição de (V.38) consiste no fato de que um mesmo ϕ_i não pode conter dois fatores $\varepsilon \in acc(v)$ e $\varepsilon' \in acc'(v)$ tais que $\varepsilon \neq \varepsilon'$. O termo β representa um fator irrestrito que não impede a EQ nos demais fatores e que possivelmente será passível de outra forma de EQ.

Observemos agora como se procede à EQ de (V.38). Em primeiro lugar, é conveniente minimizar ϕ_i aplicando-se exaustivamente as equivalências em (V.39).

[19] Em nossa implementação, utlizamos o sistema *Espresso* (Rudell, 1986), que converte para DNF qualquer função de entrada, minimizando-a.

$$\begin{aligned} \varepsilon \in A \wedge \varepsilon \in B &\equiv \varepsilon \in A \cap B \\ \varepsilon \in A \vee \varepsilon \in B &\equiv \varepsilon \in A \cup B \\ \neg(\varepsilon \in A) &\equiv \varepsilon \in \overline{A} \end{aligned} \quad \text{(V.39)}$$

Como resultado, obtém-se uma fórmula com as seguintes características:

$$\exists v\left(\beta \vee \alpha_1\phi_1 \vee \alpha_2\phi_2 \vee \ldots \vee \alpha_n\phi_n\right) \quad \boxed{\begin{array}{l} \forall i,\ 1 \le i \le n: \\ \quad \exists\{\alpha_i\}_x \wedge \\ \quad \phi_i = \varepsilon_i \in f_i(v) \end{array}} \quad \text{(V.40)}$$

Em seguida distribui-se o quantificador existencial, o que é possível devido à natureza disjuntiva da fórmula, obtendo-se (V.41).

$$\exists(\beta) \vee \exists v\left(\alpha_1\phi_1\right) \vee \exists v\left(\alpha_2\phi_2\right) \vee \ldots \vee \exists v\left(\alpha_n\phi_n\right) \quad \text{(V.41)}$$

Finalmente, basta tratar de cada fator $\exists\left(\alpha_i\phi_i\right)$ individualmente da seguinte forma:

$$\exists v\left(\alpha \wedge \varepsilon \in f(v)\right) \equiv \boxed{\varepsilon \in \bigcup_{\{\alpha\}_x}\left(f(1), f(2), \ldots, f\left(|\mathbb{V}|\right)\right)} \quad \text{(V.42)}$$

Podemos realizar um desenvolvimento análogo para o quantificador universal, como apresentado nas equações de (V.43) a (V.46). A fórmula aninhada possui, desta vez, um caráter conjuntivo, mantendo a mesma generalidade de (V.38), já que qualquer fórmula pode ser passada para a **Forma Conjuntiva Normal** (**CNF**, do inglês *Conjunctive Normal Form*)[20].

---

[20] O sistema *Espresso* também realiza conversão para CNF.

$$\forall v\big(\beta \wedge (\alpha_1 \vee \phi_1) \wedge (\alpha_2 \vee \phi_2) \wedge \ldots \wedge (\alpha_n \vee \phi_n)\big)$$
$$\boxed{\begin{array}{l}\forall i,\ 1 \le i \le n: \\ \quad \exists \{\alpha_i\}_x \wedge \\ \quad \boxed{\begin{array}{l}\exists m, k, \varepsilon, acc_1 \ldots acc_m: \\ \qquad \phi_i = \bigvee_{j=1}^{m} \big(\varepsilon \in acc_j(v)\big) \vee \bigvee_{j=1}^{m} \Big(\neg\big(\varepsilon \in acc_j(v)\big)\Big)\end{array}}\end{array}} \tag{V.43}$$

$$\forall v\big(\beta \wedge (\alpha_1 \vee \phi_1) \wedge (\alpha_2 \vee \phi_2) \wedge \ldots \wedge (\alpha_n \vee \phi_n)\big)$$
$$\boxed{\begin{array}{l}\forall i,\ 1 \le i \le n: \\ \quad \exists \{\alpha_i\}_x \wedge \\ \quad \phi_i = \varepsilon_i \in f_i(v)\end{array}} \tag{V.44}$$

$$\forall v(\beta) \wedge \forall v(\alpha_1 \vee \phi_1) \wedge \forall v(\alpha_2 \vee \phi_2) \wedge \ldots \wedge \forall v(\alpha_n \vee \phi_n) \tag{V.45}$$

$$\begin{aligned}\forall v\big(\alpha \vee \varepsilon \in f(v)\big) &\equiv \forall v\big(\neg\alpha \rightarrow \varepsilon \in f(v)\big) \\ &\equiv \forall v\Big(\neg\big(v \in \{\alpha\}_x\big) \rightarrow \varepsilon \in f(v)\Big) \\ &\equiv \forall v\Big(v \in \overline{\{\alpha\}}_x \rightarrow \varepsilon \in f(v)\Big) \\ &\equiv \boxed{\overline{\{\alpha\}}_x = \varnothing \vee \varepsilon \in \bigcap_{\{\alpha\}_x} \Big(f(1), f(2), \ldots, f\big(|\mathbb{V}|\big)\Big)}\end{aligned} \tag{V.46}$$

Como exemplo de aplicação dessa modalidade de EQ, analisemos o que acontece com o princípio **compsem**, que desenvolvemos em nossos experimentos de lexicalização, qual seja:

$$\forall v \in \mathbb{V} \left(\forall p \left(\begin{array}{l} p \in sem(v) \Leftrightarrow \\ \quad p \in bsem(v) \vee \\ \quad \boxed{\exists u \in \mathbb{V}\big(v \rightarrow_d u \wedge p \in sem(u)\big)} \end{array} \right) \right) \tag{V.47}$$

onde $bsem(v)$ e $sem(v)$ representam acessos a atributos de v modelando, respectivamente, o valor semântico básico de uma palavra e o valor semântico total da subárvore que ela domina. Aplicando (V.42) à fórmula em destaque em (V.47), obtemos (V.48) abaixo.

$$\boxed{ds(v,d) = daughters(v,d)}:$$

$$\forall v \in \mathbb{V}\left(\forall p\left(\begin{array}{l} p \in sem(v) \Leftrightarrow \\ \quad p \in bsem(v) \vee \\ \quad p \in \bigcup_{ds(v,d)}\left(sem(1), sem(2), ..., sem\left(|\mathbb{V}|\right)\right)\end{array}\right)\right) \tag{V.48}$$

Por fim, aplicando EQ básica a (V.48), chegamos ao seguinte resultado:

$$\boxed{ds(v,d) = daughters(v,d)}:$$

$$\forall v \in \mathbb{V}\left(sem(v) = bsem(v) \cup \bigcup_{ds(v,d)}\left(sem(1), sem(2), ..., sem\left(|\mathbb{V}|\right)\right)\right) \tag{V.49}$$

que reflete muito bem a idéia de que o valor semântico total de uma árvore é igual ao valor semântico básico de sua raiz mais o valor total de suas subárvores.

EQ de seleção: *oneOrMore* e *all*

Nossa segunda classe de fórmulas de interesse sujeitas a EQ de seleção está caracterizada abaixo:

$$\left\{oneOrMore\left(\left(\alpha_1 \cap \beta_1(v)\right) \cup \left(\alpha_2 \cap \beta_2(v)\right) \cup ... \cup \left(\alpha_n \cap \beta_n(v)\right)\right)\right\}_v$$

$$\boxed{\begin{array}{l} \forall i,\ 1 \le i \le n: \\ \quad \alpha_1 \text{ livre de } v \wedge \\ \quad \boxed{\begin{array}{l} \exists m, k, \varepsilon, grv_1 ... grv_m : \\ \qquad \beta_i(v) = \bigcap_{j=1}^{k}\left(grv_j(v)\right) \cap \bigcap_{j=1}^{m}\left(\overline{grv_j(v)}\right) \end{array}} \end{array}} \tag{V.50}$$

onde todo $grv_j(v)$ consiste numa variável gráfica qualquer do vértice v — como $daughters(v, d_j)$ ou $upEndL(v, l_j, d_j)$, para alguma dimensão d_j e rótulo de aresta l_j. Como acontecia com a classe anterior, no tocante à estrutura imposta pelos operadores, (V.50) é

bastante geral, já que toda fórmula de conjuntos pode ser passada para uma forma análoga à DNF, que chamamos de **SDNF** (do inglês *Set Disjunctive Normal Form*)[21]. A grande diferença, desta vez, é que estamos estendendo a definição de $\{\phi\}_x$ para tratar fórmulas geradas por EQs básicas, permitindo encontrar uma expressão PLE para algo como:

$$\left\{v : oneOrMore\left(\left(\alpha_1 \cap \beta_1(v)\right) \cup \left(\alpha_2 \cap \beta_2(v)\right) \cup \ldots \cup \left(\alpha_n \cap \beta_n(v)\right)\right)\right\} \qquad \text{(V.51)}$$

Em outras palavras, estamos permitindo que EQs básicas também envolvam restrições de seleção.

Aplicando-se as operações preexistentes de $\{\phi\}_x$ a (V.50), obtém-se o seguinte:

$$\left\{\bigvee_{i=1}^{n} oneOrMore\left(\alpha_i \cap \beta_i(v)\right)\right\}_v = \bigcup_{i=1}^{n} \left\{oneOrMore\left(\alpha_i \cap \beta_i(v)\right)\right\}_v \qquad \text{(V.52)}$$

Basta agora tratar de cada fator $\left\{oneOrMore\left(\alpha_i \cap \beta_i(v)\right)\right\}_v$ individualmente, como mostrado abaixo:

$$\left\{oneOrMore\left(\alpha \cap \beta(v)\right)\right\}_v = \bigcup_{\alpha}\left(\widetilde{\beta(1)}, \widetilde{\beta(2)}, \ldots, \widetilde{\beta\left(|\mathbb{V}|\right)}\right) \qquad \text{(V.53)}$$

onde $\widetilde{\varepsilon}$ é a **inversão do ponto de vista** da expressão ε e é definida pelas seguintes regras de cálculo:

---

[21] Mais uma vez, o *Espresso* pode ser usado para gerar uma tal forma minimizada, por isomorfia entre expressões lógicas e de conjuntos.

$$
\begin{aligned}
\widetilde{\alpha \odot \beta} &= \widetilde{\alpha} \odot \widetilde{\beta} \quad \left(\odot \in \{\cup, \cap\}\right) \\
\widetilde{\overline{\beta}} &= \overline{\widetilde{\beta}} \\
\widetilde{eq(v,d)} &= eq(v,d) \\
\widetilde{mothers(v,d)} &= daughters(v,d) \\
\widetilde{daughters(v,d)} &= mothers(v,d) \\
\widetilde{mothersL(v,l,d)} &= daughtersL(v,l,d) \\
\widetilde{daughtersL(v,l,d)} &= mothersL(v,l,d) \\
\widetilde{up(v,d)} &= down(v,d) \\
\widetilde{down(v,d)} &= up(v,d) \\
\widetilde{upEndL(v,l,d)} &= downL(v,l,d) \\
\widetilde{downL(v,l,d)} &= upEndL(v,l,d)
\end{aligned}
\qquad (V.54)
$$

Podemos realizar um desenvolvimento análogo para o predicado *all*, como apresentado nas equações de (V.55) a (V.57). O operando possui, desta vez, um caráter intersectivo, mantendo a mesma generalidade de (V.50), já que qualquer expressão de conjuntos pode ser passada para uma forma análoga à CNF, que chamamos de **SCNF** (do inglês *Set Conjunctive Normal Form*).

$$
\left\{ all\left(\left(\alpha_1 \cup \beta_1(v)\right) \cap \left(\alpha_2 \cup \beta_2(v)\right) \cap \ldots \cap \left(\alpha_n \cup \beta_n(v)\right) \right) \right\}_v
\boxed{
\begin{array}{l}
\forall i,\ 1 \le i \le n: \\
\quad \alpha_1 \text{ livre de } v \wedge \\
\quad \boxed{
\begin{array}{l}
\exists m, k, \varepsilon, grv_1 \ldots grv_m: \\
\qquad \beta_i(v) = \bigcup_{j=1}^{k} \left(grv_j(v) \right) \cup \bigcup_{j=1}^{k} \left(\overline{grv_j(v)} \right)
\end{array}
}
\end{array}
}
\qquad (V.55)
$$

$$
\left\{ \bigwedge_{i=1}^{n} all\left(\alpha_i \cup \beta_i(v)\right) \right\}_v = \bigcap_{i=1}^{n} \left\{ all\left(\alpha_i \cup \beta_i(v)\right) \right\}_v \qquad (V.56)
$$

$$
\left\{ all\left(\alpha \cup \beta(v)\right) \right\}_v = \bigcap_{\overline{\alpha}} \left(\widetilde{\beta(1)}, \widetilde{\beta(2)}, \ldots, \widetilde{\beta\left(|\mathbb{V}|\right)} \right) \qquad (V.57)
$$

Como exemplo de aplicação dessa modalidade de EQ, analisemos como a seguinte fórmula seria processada:

$$\forall v,u \in \mathbb{V}\begin{pmatrix} u \rightarrow_a v \Leftrightarrow \\ \quad u \rightarrow_b v \wedge \\ \quad \neg\exists w \in \mathbb{V}\left(w \rightarrow_b v \wedge w \lhd_c u\right) \end{pmatrix} \tag{V.58}$$

Trata-se do princípio ***topmothers***, desenvolvido em nossos experimentos e que impõe que, desconsiderando-se rótulos de aresta, a projeção em a seja como que uma cópia da projeção em b, salvo que, para todo vértice v, apagam-se as arestas de todos os pais de v "não-mínimos" segundo uma relação de ordem modelada em c. Em específico, se $b = c$, então *topmothers* impõe que a projeção em a seja a árvore encontrada em b tal que, para todo $v \in \mathbb{V}$, v mantenha como pai em a apenas o pai mais próximo da raiz em b.

Aplicando EQ básica, a (V.58) temos o seguinte resultado:

$$\forall v,u \in \mathbb{V}\begin{pmatrix} u \rightarrow_a v \Leftrightarrow \\ \quad u \rightarrow_b v \wedge \\ \quad \neg oneOrMore\left(mothers(v,b) \cap up(u,c)\right) \end{pmatrix} \tag{V.59}$$

Por fim, eliminando totalmente u de (V.59) por EQ básica, obtemos (V.60).

$$\boxed{\begin{array}{l} ms(v,d) = mothers(v,d) \\ dw(v,d) = down(v,d) \end{array}}:$$

$$\forall v \in \mathbb{V}\left(ms(v,a) = ms(v,b) \cap \overline{\bigcup_{ms(v,b)} \left(dw(1,c), dw(2,c), \ldots, dw\left(|\mathbb{V}|, c\right)\right)}\right)$$
$$\equiv \tag{V.60}$$
$$\forall v \in \mathbb{V}\left(ms(v,a) = ms(v,b) - \bigcup_{ms(v,b)} \left(dw(1,c), dw(2,c), \ldots, dw\left(|\mathbb{V}|, c\right)\right)\right)$$

V.1.2 Eliminação de subexpressões comuns

Uma otimização importante realizada pelo QO é variante da clássica **Eliminação de Subexpressões Comuns** (**CSE,** do inglês *Common Subexpression Elimination*), que é especialmente bem-vinda numa linguagem como a PL, em que simples expressões podem ter complexidade polinomial no tamanho da entrada/número de vértices. Tome, por exemplo, a fórmula PL abaixo:

```
forall V: forall L:
   V ~= 1 =>
      (L in V.ORD.entry.after | L = start =>
         forall V1: edge(V V1 L D) => edge(V V1 ORD)) &
      (~(L in V.ORD.entry.after | L = start) =>
         forall V1: edge(V V1 L D) => edge(V1 V ORD))
```
(V.61)

De (V.61), deriva-se por EQ a seguinte expressão PLE:

```
forall V: forall L:
   [V ~= 1]_A =>
      ([[L in V.D.entry.after]_B | [L = start]_C]_D =>
         [daughters(V L D)]_E subseteq daughters(V ORD)) &
      (~([[L in V.D.entry.after]_F | [L = start]_G]_H) =>
         [daughters(V L D)]_I subseteq mothers(V ORD))
```
(V.62)

Em (V.62), destacam-se nove (sub)expressões com propriedades notáveis, identificadas pelas letras de *A* a *I*. Em primeiro lugar, vale notar que existem quatro pares de expressões iguais, quais sejam $B = F$, $C = G$, $D = H$ e $E = I$, os quais representariam trabalho replicado caso o PW usasse (V.62) tal como está para gerar código-objeto. Outra fonte de desperdício de recursos é a expressão *A*, desta vez não por estar repetida, mas por estar mal posicionada. Para cada vértice, *A* será avaliada $|\mathbb{U}_{\mathtt{L}}|$ vezes devido à expansão de **`forall L`**. Felizmente, por meio de seu módulo de CSE, o QO converte (V.62) na seguinte forma de semântica declarativa equivalente:

```
forall V:
let Auto1 = V ~= 1:
   forall L:
   let Auto2 = L in V.ORD.entry.after | L = start:
      Auto1 [=>]
         (Auto2 => daughters(V L D) subseteq mothers(V ORD)) &
         (~Auto2 => daughters(V L D) subseteq mothers(V ORD))
```
(V.63)

Os constructos **let** encontrados em (V.63) calculam dois *resultados parciais* — de *A* e *D/H* — e os associam a variáveis — **Auto1** e **Auto2** — por meio das quais podem ser aproveitados nas expressões aninhadas. Note que, na transformação de (V.62) em (V.63), o QO reposicionou a expressão *A* devidamente e foi sensível ao fato de que, ao tratar de *D* e *H*, as demais repetições deixaram de existir, evitando a geração de variáveis em demasia. Além disso, deixou intocadas as expressões *E* e *I*, pois sua avaliação tem custo computacional constante e irrisório, provavelmente menor mesmo que o incorrido por **let**. Tais expressões são ditas **triviais**.

Sobre a semântica operacional de **let**, vale ainda observar que os resultados parciais são calculados sob um regime de *avaliação preguiçosa*, isto é, dada uma expressão do tipo:

$$\texttt{let}\ v\ \texttt{=}\ \varepsilon\ \texttt{:}\ \phi$$

garante-se que, na avaliação de ϕ, não só a expressão ε não será avaliada mais de uma vez, mas também poderá simplesmente não ser avaliada, caso os propagadores responsáveis por ϕ prescindam do valor de ε para calcular o valor-verdade de ϕ. No nosso exemplo, isso aconteceria, entre outros, para **V** igual a 1, ou seja, **Auto1** falso. Nesse caso, o propagador *P* gerado pelo operador de implicação destacado em (V.63) requisitaria a avaliação de **Auto1**, esperaria pelo estatuto do resultado (determinado ou não) e, visto ser este determinado e falso, *P* já acusaria a validade de sua implicação, sem jamais chegar a avaliar o operando direito ou, por conseguinte, **Auto2**. Devido a esse regime de avaliação, as variáveis declaradas por **let** são ditas **preguiçosas**.

Em linhas gerais, o algoritmo de CSE utilizado pelo QO realiza os seguintes passos sobre uma dada árvore PLAE:

- promove-se um renomeamento de variáveis tal que toda expressão de quantificação $Q\, v(\phi)$ use um identificador v único no seu escopo. Na fórmula resultante, nenhuma nova variável jamais obscurece uma variável em escopo;

- processa-se a árvore de maneira *top-down*. Seja ε a subárvore/subexpressão corrente; *qvs*, a lista das variáveis declaradas pelos nós ascendentes de ε, em ordem crescente de distância a ε; e *deps*, o conjunto de variáveis que ocorrem em ε. Se ε for não-trivial, substitui-se-a pela expressão **ctx(***ctxv* ε' *req***)**, onde: (i) *ctxv* é a primeira variável em *qvs* pertencente a *deps*, consistindo na variável do quantificador acima do qual ε perde o sentido; (ii) *req* é verdadeiro se e só se *ctxv* não for o primeiro elemento de *qvs*, sinalizando que ε deve ser reposicionado; e ε' é a expressão resultante da recursão desse processamento sobre ε;

- processa-se a árvore de maneira *bottom-up*. Seja ε a subárvore/subexpressão corrente. Se ε for uma expressão de quantificação declarando uma variável *curv* e tendo ϕ como fórmula aninhada, então se faz o seguinte:

    - cria-se uma lista vazia $D:\left[PLAE \times Id \times Bool\right]$, cujos elementos associam uma fórmula PLAE, um identificador de variável e um valor-verdade. Assumam-se as seguintes definições auxiliares:

        $getDef\left((\varepsilon'', _, _)\right) = \varepsilon''$,

        $getId\left((_, id, _)\right) = id$,

        $getReq\left((_, _, req)\right) = req$,

        $lookup(\varepsilon'') =$ a primeira tupla t de D tal que $getDef(t) = \varepsilon''$;

- processa-se ϕ de maneira *bottom-up*. Seja ε' a subárvore/subexpressão corrente. Se ε' for do tipo **`ctx(`**$curv\ \ \varepsilon''\ \ req$**`)`**, então:
  - se já existe $lookup(\varepsilon'')$, então se procede à atualização $getReq\big(lookup(\varepsilon'')\big) \leftarrow \top$, indicando que há repetição de ε''; senão, insere-se $(\varepsilon'', newid, req)$ como cabeça de *D*, onde $newid$ é um novo identificador de variável;
  - substitui-se ε' por $getId\big(lookup(\varepsilon'')\big)$, ou seja, o identificador da variável, seja preexistente ou nova;
- processando-se *D* de trás para frente, para toda entrada $t = (\varepsilon'', id, req)$ tal que $req = \perp$, substitui-se *id* por ε'' em $getDef(t')$, para toda t' que preceda *t* em *D*, e elimina-se *t* de *D*. Se *t* for o primeiro elemento de *D*, ε'' deve substituir ϕ;
- sendo $D = \big[(\varepsilon''_1, id_1, \top), (\varepsilon''_2, id_2, \top), \ldots, (\varepsilon''_n, id_n, \top)\big]$, atualiza-se ϕ com uma cadeia de expressões **`let`** da seguinte forma:

$$\phi \leftarrow \big(\texttt{let}\ id_n = \varepsilon''_n :\ \texttt{let}\ id_{n-1} = \varepsilon''_{n-1} :\ \ldots\ \texttt{let}\ id_1 = \varepsilon''_1 :\ \phi\big);$$

- por fim, processa-se a árvore de maneira *top-down*, expandindo-se toda variável *v* tal que (i) *v* tenha um único ponto de aplicação no código e (ii) não haja nenhum quantificador entre esse ponto e a declaração de *v*. Tais **variáveis espúrias** surgem em virtude de expressões reposicionadas, cujas subexpressões são também reposicionadas e terão *req* sempre verdadeiro.

V.2 Avaliação

A eficácia das otimizações do QO foi avaliada de forma empírica como descrevemos a seguir. Vale notar, em primeiro lugar, que a EQ de seleção ainda não é implementada pelo QO e que mesmo a EQ básica só foi aplicada a variáveis-vértice, já que a eliminação de outros tipos de variável — em especial, rótulos de aresta — requer EQ de seleção.

O experimento foi realizado num computador pessoal com processador Pentium Core 2 Duo T7200 a 2,0GHz, com 2,0Gb de RAM, e consistiu em quatro rodadas, cada qual composta dos seguintes passos:

- compilar todos os princípios da gramática **`dissPW.ul`**, que conta exclusivamente com princípios desenvolvidos no PW, utilizando uma configuração de otimização específica para a rodada;
- recompilar o XDK para esses novos princípios, alimentá-lo com **`dissPW.ul`** e executar o *parsing* de todas as 60 sentenças-exemplo distribuídas com a gramática, anotando, para cada uma, o tempo necessário para encontrar todas as suas possíveis análises. Por precaução, cada sentença foi submetida duas vezes ao XDK e considerou-se a média dos dois tempos observados;

As quatro diferentes configurações de otimização utilizadas foram:

a) todas as otimizações desabilitadas;

b) todas as otimizações habilitadas, o que abrange (i) EQ, (ii) CSE e (iii) avaliação preguiçosa das variáveis geradas em (ii);

c) apenas (iii) desabilitada, ou seja, todas as variáveis geradas pela CSE são avaliadas no ponto de declaração;

d) CSE desabilitada, o que anula (iii), ou seja, apenas EQ está operante.

As sentenças-exemplo têm de 4 a 16 *tokens*, gerando análises com o mesmo número de vértices. O comprimento médio das sentenças é 7.95, com desvio-padrão 2.57.

Os dados observados foram analisados da seguinte forma. O parâmetro de avaliação aplicado foi o **ganho** em tempo proporcionado pelas configurações (b), (c) e (d) — ditas **otimizadas** — contra (a) — dita ***baseline***. Para cada sentença-exemplo *S* e configuração otimizada *C*, o ganho foi calculado como a simples razão:

$$ganho_C(S) = \frac{t_C(S)}{t_0(S)}$$

onde $t_C(S)$ e $t_0(S)$ são os tempos observados no processamento de *S* por *C* e pela *baseline*, respectivamente.

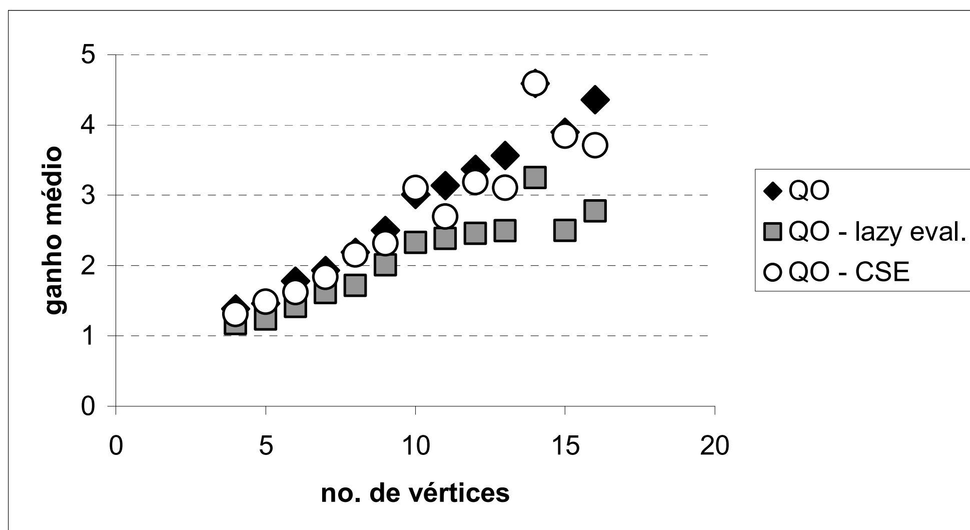

Figura 25: ganho médio em tempo em função do número de vértices para diferentes configurações de otimização

Os resultados obtidos são muito positivos e estão bem representados pelo gráfico da Figura 25. Ali se apresenta, para cada configuração otimizada e cada comprimento *n* observado na entrada, o ganho médio $M_C(n)$ assim calculado:

$$exs(n) = \left\{ S : S \in Exemplos \wedge |S| = n \right\}:$$
$$M_C(n) = \frac{\sum_{S \in exs(n)} ganho_C(S)}{|exs(n)|}$$

onde *Exemplos* é o córpus de sentenças-exemplo, e $|S|$ é o comprimento da sentença *S*.

O gráfico aponta para as seguintes conclusões:

- que a configuração completa é a que mais ganho obtém em média;
- que a EQ é responsável por uma queda real na complexidade de tempo, visto que o ganho cresce com *n* em todas as configurações;
- que a CSE é benéfica, mas não deve seguir um regime de avaliação gulosa, já que a configuração (c) tem desempenho melhor que (b);
- que muitos valores devem ficar determinados antes da execução de certos operadores lógicos, o que permite que (c) consiga, apesar de não usar CSE, conseguir melhor desempenho que (b), evitando por completo a avaliação de certas expressões.

A título de curiosidade, apresentamos também um gráfico do ganho mínimo contra o tamanho da entrada, na Figura 26, que corrobora a figura anterior. Nele se destaca a única ocorrência de ganho menor que um, ou seja, uma perda, cometida pela configuração (b). Em todo o experimento, o tempo máximo despendido para uma sentença (n = 16) foi de 26,5s (pelo *baseline*) contra 6,1s (versão completa do QO). O ganho máximo observado foi 4.59 (idem), para uma sentença de 14 palavras.

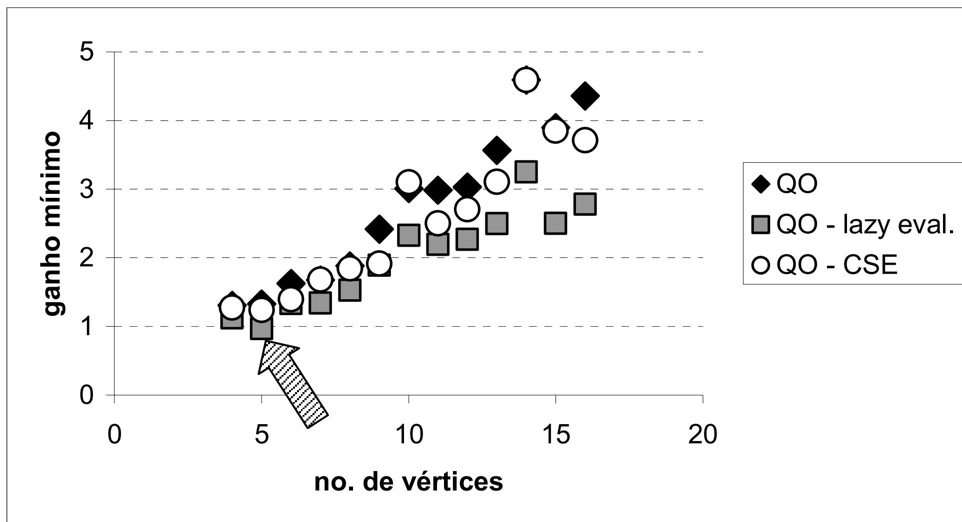

Figura 26: dispersão do ganho mínimo em função do número de vértices para diferentes configurações de otimização

VI Capturando a Disjunção na Lexicalização com a XDG

Recapitulando, o termo *lexicalização* (Reiter e Dale, 2000) aplica-se à decisão de quais dentre um conjunto de itens lexicais potencialmente aplicáveis, todos realizando ao menos parte do significado pretendido, irão realmente fazer parte do texto gerado. Ela é tida como uma tarefa de RL necessária – especialmente se levarmos em consideração que o termo *tarefa* não necessariamente implica *pipelining* – e notavelmente importante. Por exemplo, mesmo que a realização de um sintagma como "*a ballerina*[22]" deva muito à geração de expressões referenciais, uma tarefa complementar, é ainda uma questão de lexicalização preferir ou não esse sintagma específico a alternativas igualmente legítimas e possíveis, tais como "*a female dancer*", "*a dancing woman*", "*a dancing female person*". Contudo, antes da formulação de critérios de priorização ou preferências de seleção e mesmo como substrato para tanto, a questão fundamental da lexicalização é exatamente alternância, escolha — em uma palavra, *disjunção*.

Dada a natureza combinatória da linguagem e especificamente a intercambialidade entre itens lexicais, culminando em inúmeras soluções válidas para uma mesma tarefa de lexicalização, a disjunção pode bem se tornar uma fonte considerável de complexidade combinatória. O assunto central neste capítulo é unicamente a disjunção na lexicalização como base para modelos de lexicalização mais avançados, e nosso propósito é precisamente descrever um modelo XDG que (i) *capture o potencial disjuntivo da lexicalização*, isto é, permita a geração de todas as paráfrases que sejam solução para a tarefa (de acordo com um dado modelo de língua), (ii) *assegure boa-formação*, especialmente evitando **sobre-redundância** (tal como a encontrada em "**a dancing female dancer/ballerina/woman*) e anomalias sintáticas ("**a wo-*

---

[22] Todos os exemplos deste capítulo estão em inglês, que foi a língua usada em nossos experimentos.

man dancer"), e o faça de maneira *modular* e *eficiente*, tendo uma implementação que permita forte propagação e, portanto, propensa a manter a complexidade em níveis aceitáveis.

VI.1 Modelagem

A entrada da Realização Lingüística

Num contexto de *parsing*, o tipo de entrada é bastante trivial, notadamente sentenças escritas ou possivelmente outras unidades de texto. A criação de modelo é também muito simples nesse caso e consiste em (i) criar exatamente um vértice para cada *token* de entrada, (ii) fazer com que cada vértice selecione uma dentre as entradas lexicais indexadas por seu respectivo *token*, (iii) impor restrições automaticamente geradas a partir dos princípios encontrados na definição de gramática e (iv) atribuir deterministicamente valores aos atributos relativos à ordem linear dos vértices de modo a refletir a ordem real dos *tokens* na entrada.

No que concerne à geração, as coisas não são tão claras. Primeiramente, tome-se como exemplo a entrada, que usualmente varia de acordo com as aplicações e sistemas, sem mencionar o fato de que a representabilidade e a computabilidade do significado em geral são questões em aberto. A criação de modelo não deve ficar atrás, já que ela é uma função direta da entrada. Contudo, podemos, em alguma medida, afirmar o que *não é* entrada da geração. Sob a hipótese de um sistema de geração baseado na XDG realizando lexicalização, não é provável que a entrada contenha alguma representação direta de projeções em PA completamente especificadas, muito embora essa seja considerada uma saída satisfatória para um sistema de análise (!). O que ocorre aqui é que gerar uma projeção em PA de entrada pressuporia que a lexicalização já tivesse sido realizada. Em outras palavras, projeções em PA dando conta de, por exemplo, "*a ballerina*" e "*a dancing female person*" não têm absolutamente nenhuma relação uma com a outra, enquanto o nosso intuito é exatamente fornecer uma entrada única que possibilite ambas as realizações. Por isso, projeções em PA são elas mesmas parte da saída da geração e

são aceitáveis como saída do *parsing* na medida em que "de-lexicalização" possa ser considerada uma tarefa trivial, o que não é necessariamente verdade.

Com base nas razões acima, parece razoável decompor os predicados em PA em predicados mais simples, primitivos, que revelem suas inter-relações. Para os propósitos da presente discussão, entendemos que baste especificar que nossa entrada conterá conjunções de lógica de primeira ordem *flat* como a seguinte:

$$\exists x \left(dance(x) \wedge female(x) \wedge human(x) \right) \qquad \text{(VI.1)}$$

a fim de caracterizar entidades, embora esteja claro que a linguagem de entrada de um sistema de geração completo certamente deva ter um componente lógico mais estrito que a lógica de primeira ordem e provavelmente envolva cruzamentos ainda com outros formalismos. Os predicados primitivos, felizmente, não precisam ser necessariamente unários; e, por exemplo, "*a ballerina tapped a lovely she-dog*", poderia ser gerada a partir da seguinte entrada

$$\exists e, x, y \begin{pmatrix} dance(x) \wedge female(x) \wedge \\ \wedge human(x) \wedge event(e) \wedge \\ \wedge past(e) \wedge tap(e, x, y) \wedge \\ \wedge female(y) \wedge dog(y) \wedge \\ lovely(y) \end{pmatrix} \qquad \text{(VI.2)}$$

Deleção como substância da disjunção

Naturalmente, a simples criação de um vértice para cada literal semântico da entrada não é de modo algum a idéia por trás de nosso modelo. Por exemplo, se "*woman*" chegar a ser realmente empregada em uma tarefa de lexicalização específica, então deverá continuar figurando como um único vértice nas análises XDG, apesar de potencialmente cobrir um complexo de literais. Com efeito, a XDG e, em específico, projeções em PA, devem se comportar e parecer como no *parsing*, na medida do possível.

No entanto, uma diferença considerável das análises em nosso modelo de geração em comparação com o *parsing* está no papel e escopo da deleção, que constitui, agora, a própria substância da disjunção. Ao atribuir a todos os vértices, salvo a raiz da análise, uma entrada lexical sincronizando deleção em todas as dimensões, construímos uma forma irrestrita de disjunção por meio da qual conjuntos inteiros de vértices podem agir como se não fizessem parte da solução. Agora é possível criar vértices à vontade — mesmo um para cada item lexical aplicável — e contar com o fato de que, muitas saídas malformadas que o conjunto de todas as soluções contenha, ele ainda cobre todas as paráfrases corretas, isto é, aquelas em que todos e apenas os vértices corretos tenham sido deletados. Por exemplo, se um vértice for criado para cada um dos itens "*ballerina*", "*woman*", "*dancer*", "*dancing*", "*female*" e "*person*", então é certo que todas as combinações dessas palavras, inclusive as corretas, serão geradas.

Nosso esquema obviamente precisa de restrições adicionais; contudo, já deve estar aparente a idéia geral de que pretendemos finalizar a criação de modelo — ou melhor, começar a busca — com (i) um conjunto de vértices perfeitamente flutuantes, na medida em que nenhuma aresta é dada nesse momento, todos igualmente dispostos a serem deletados — e freqüentemente o sendo — e (ii) um conjunto de restrições para excluir saídas mal-formadas e propiciar eficiência. Há duas lacunas principais neste resumo, quais sejam:

- quais são estas restrições; e
- como exatamente estes vértices são criados.

Restringimo-nos aqui à primeira questão. A segunda envolve questões além da lexicalização, permeando todas as tarefas de geração, e é ainda objeto de pesquisa. Conseqüentemente, em todos os nossos experimentos, uma parte da criação de modelo foi feita manualmente.

Uma parte das novas restrições que definiremos requer uma visão de PA em que os vértices deletados deixem de existir mesmo como filhos da raiz da análise. Para isso, postulamos uma nova dimensão DPA, que se relaciona com PA por meio do seguinte princípio:

```
defprinciple "principle.del" {
   dims {D DELD}
constraints {
forall V: forall L: forall U:
   (L ~= del =>
      edge(V U L D) <=> edge(V U L DELD))
   & (L = del =>
      edge(V U L D) <=>
         ~exists W: edge(W U DELD) | edge(U W DELD))
}
```

(VI.3)

Como os vértices se relacionam

Na descrição que segue, ater-nos-emos principalmente ao que é novo em nosso modelo em comparação à prática usual em modelagem XDG. Para tanto, enfatizaremos a dimensão PA e as novas restrições que tivemos que introduzir a fim de garantir que apenas as projeções em PA desejadas emerjam. Nosso modelo recorre a uma dimensão que já foi utilizada por outras gramáticas, mas não surge especificamente em **diss.ul** e pode ser, por isso, novidade para o leitor. Trata-se de DS (*Deep Syntax*), que intermedeia PA e ID e, *grosso modo*, captura ID antes de fenômenos de alçamento. Salvo por poucas observações sobre DS e sua relação com PA, assumimos sem mais comentários a concorrência de outras dimensões, princípios e conceitos da XDG em qualquer aplicação real de nosso modelo.

Referentes, argumentos e assim os vértices se encontram

Em grande parte, eliminar saídas mal-formadas implica impor restrições sobre arestas aceitáveis, especialmente quando se considera que tudo o que temos para começar é alguns vértices flutuantes. Recordemos que o assunto de PA é, por excelência, argumentos de predicados, que são necessariamente variáveis graças à natureza *flat* de nossas entradas semânticas. Cada ares-

ta de PA relaciona um predicado com um de seus argumentos e, portanto, "diz respeito" a uma única variável. Por isso, nossa primeira preocupação deve ser garantir que cada aresta de PA incida em um vértice que também "diga respeito" à mesma variável que a aresta.

A fim de fornecer subsídio para uma tal concordância de "dizer a respeito", por assim dizer, deve-se primeiro contemplar o "dizer a respeito". Desse modo, postulamos que cada vértice deve ter agora dois novos atributos, a saber (i) *hook*, identificando o referente do vértice, i.e., a variável a que diz respeito, e (ii) *holes*, mapeando cada rótulo *l* de aresta de PA no argumento (uma variável), ao qual toda aresta de saída etiquetada com *l* deve dizer respeito. Normalmente, estes atributos devem ser lexicalizados. A coincidência com a terminologia de Copestake et al. (Copestake et al., 2001) não é casual; na verdade, nossa formulação pode ser vista como um fragmento modular da deles, visto que nem nossos *holes* envolvem atributos sintáticos, nem questões de escopo são jamais mencionadas. Como de hábito na XDG, delegam-se a outros módulos as tarefas de lidar com escopo lógico e relacionar argumentos semânticos e papéis sintáticos. O papel destes novos atributos é ilustrado pela Figura 27, na qual uma seta não significa uma aresta, mas a possibilidade de estabelecimentos de arestas.

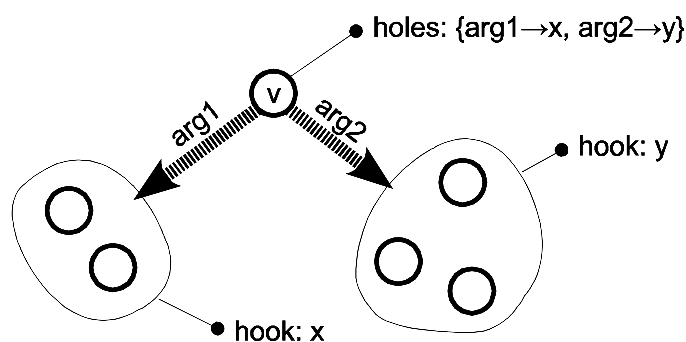

Figura 27: o papel dos atributos *hook* e *holes*

Essa restrição se materializa numa gramática XDG por meio do princípio **`hooksAndHoles`**, listado abaixo, aplicado a DPA:

```
defprinciple "principle.hooksAndHoles" {
   dims {PA DPA}
constraints {
forall V: forall L: forall U:
   edge(V U L DPA)
   <=>
   [L U.PA.entry.hook] in V.PA.entry.holes
}
```

(VI.4)

Completude e composicionalidade semântica

Em seguida, procuraremos assegurar a **completude**, isto é, que toda solução deve transmitir todo o conteúdo semântico pretendido. Para esse fim, os vértices devem ter atributos que envolvam informação semântica, a mais simples das quais é ***bsem***, que representa o **conteúdo semântico de base**, ou antes, a contribuição que uma entrada lexical pode fazer sozinha para o significado do todo. Por exemplo, pode-se dizer que "*woman*" contribua com $\lambda x.female(x) \wedge human(x)$, enquanto "*female*", apenas $\lambda x.female(x)$. Normalmente, *bsem* deve ser um atributo lexical e não passa de um conjunto de inteiros, cada qual associado biunivocamente a um literal semântico.

Além disso, postulamos o atributo *sem* para conter o conteúdo semântico atual de cada vértice, que não deve ser lexical, mas sim calculado através de `compsem`, um princípio impondo composicionalidade semântica que já utilizamos como exemplo no capítulo anterior (pág. 109). A expressão PL desse princípio é a seguinte:

```
defprinciple "principle.compsem" {
   dims {PA DPA}
constraints {
forall V: forall P:
   P in V.PA.attr.sem
   <=>
   P in V.PA.entry.bsem |
      (exists U: edge(V U DPA) & P in U.PA.attr.sem)
}
```

(VI.5)

Finalmente, a completude é imposta por meio do atributo ***axiom*** do vértice, sobre o qual opera o seguinte princípio:

```
defprinciple "principle.axiom" {
   dims {PA}
constraints {
forall V: forall P:
   P in V.PA.entry.axiom => P in V.PA.attr.sem
}
```

(VI.6)

A idéia é termos *axiom* como um atributo lexical e atribuir-lhe consistentemente a constante neutralizadora ∅ para todas as entradas lexicais, exceto as da raiz da análise, caso em que *axiom* deve equivaler ao conteúdo semântico pretendido.

Classes de co-referência, concentradores e revisão de PA e DS

O impedimento inevitável à propagação é a escolha intrínseca, isto é, aquela entre coisas equivalentes, a qual desejamos conservar. É exatamente isso que desejamos capturar para a lexicalização, enquanto tentamos tornar disponível a maior quantidade possível de determinação para minimizar falhas. Para isso, nossa estratégia é simplificar as projeções em PA, com vértices co-referentes — isto é, tendo todos o mesmo referente ou *hook* — organizados em feixes ao redor, ou antes, *diretamente abaixo* de um único vértice, o assim chamado **concentrador**. Isso oferece diversas vantagens, a saber:

1. O número de vértices-folha é maximizado, cujos atributos *sem* são determinados e equivalem a seus respectivos *bsem*;

2. vértices co-referentes tendem ou a ser potenciais irmãos abaixo de um concentrador ou a ser deletados. Isso permite que a maioria das restrições seja expressa em termos de relações de filiação (arestas diretas) ou de irmandade. Tal proximidade e concentração são bastante oportunas, porque estamos lidando simplesmente com relações *potenciais*, já que os vértices serão usualmente deletados. Em outras palavras, nossas restrições objetivam

principalmente eliminar relações indesejadas, em vez de estabelecer relações corretas. Esta última ação deve permanecer uma questão de escolha.

Para converter relações de irmandade em DPA em arestas diretas, postulamos a dimensão DPASIS, relacionada com DPA por meio do seguinte princípio:

```
defprinciple "principle.sisters" {
   dims {D DSIS}
constraints {
   forall V: forall U: forall L:
      edge(V U L SISTERS)                              (VI.7)
      <=>
      V ~= U &
         (exists W: edge(W V D) & edge(W U L D))
}
```

Cumpre definir agora quais sejam os melhores candidatos a concentradores. Ter concentradores diferentes em realizações equivalentes e alternativas, tais como "*a* *ballerina*", "*a female* *dancer*" ou "*a dancing* *woman*" (concentradores hipotéticos sublinhados), seria bastante prejudicial, uma vez que a tarefa de determinar a "concentradoridade" estaria, então, fatalmente ligada à própria disjunção da lexicalização e pouca determinação poderia ser derivada antes de se comprometer com essa ou aquela realização. Em face disso, o candidato natural deve ser algo que se mantenha constante, notadamente o *determinante*.

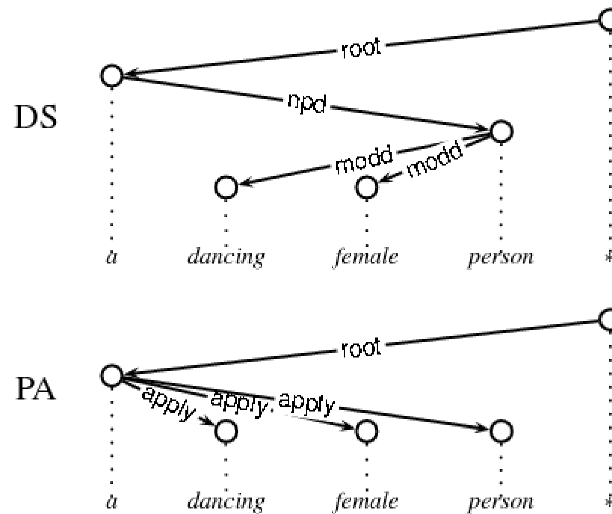

Figura 28: novas análises em PA e DS para "*a dancing female person*"

Eleger artigos para concentradores significa que eles agora dominam diretamente seus respectivos substantivos e modificadores na dimensão em PA, como mostra a Figura 28 para "*a dancing female person*". Um novo rótulo de aresta **`apply`** é postulado para conectar concentradores com seus complementos, envolvendo as seguintes invariantes:

1. para todo vértice *v*, $hook(v) = holes(v)(apply)$, isto é, apenas vértices co-referentes são ligados por arestas *apply*;

2. toda entrada lexical de concentrador define uma valência permitindo qualquer número de arestas *apply* de saída, mas requerendo pelo menos uma.

Grosso modo, a intuição por trás desse novo esquema para PA é que a ocorrência de uma palavra lexical (em oposição a "gramatical") corresponde à avaliação de uma expressão lambda, resultando em um novo predicado unário construído a partir do *bsem* da palavra/vértice e dos *sem* de seus filhos. Por sua vez, toda aresta *apply* denota a aplicação de um tal predicado à/ao variável/referente de um concentrador. De fato, mesmo verbos podem ser tratados analogamente, se constituintes *Infl* forem modelados, constituindo os concentradores das formas básicas dos verbos. Também é útil a intuição de que PA abstrai a maioria das oposições mor-

fossintáticas, tal como as que existem entre substantivos e adjetivos. A subordinação da última classe de palavras torna-se um fenômeno estritamente sintático ou, em todo caso, assunto de outras dimensões.

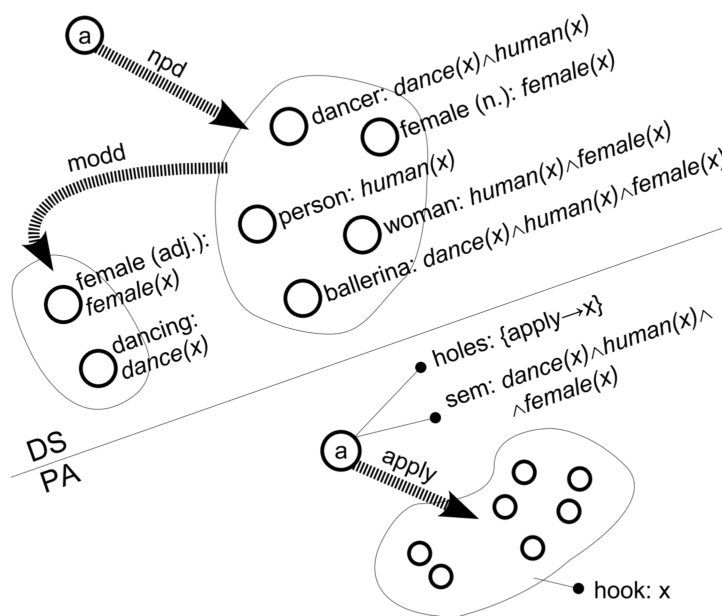

Figura 29: condições de início com vértices perfeitamente flutuantes na lexicalização de "*a ballerina*" e suas paráfrases

Por outro lado, tais oposições importam muito a DS, que deve se manter em grande parte como antes, a não ser pelo fato de manter a dominância do concentrador, o que simplifica o esquema sobremaneira. Como resultado, o artigo deve permanecer como núcleo do sintagma nominal, ou melhor, determinante, o que não é uma abordagem sem precedentes, apenas inédita na XDG. Naturalmente, estruturas sintáticas padrão devem aparecer abaixo dos determinantes, como exemplificado na Figura 29. Com base nisso, o achatamento de PA e sua relação com DS podem ser realizadas diretamente pelos seguintes princípios do XDK: **`climbing`**, por meio do qual PA é restringida a ser uma versão achatada de DS, e **`barriers`**, que faz com que os concentradores bloqueiem o alçamento/achatamento por meio de atributos lexicais especiais. A Figura 29 ilustra as condições iniciais para a lexicalização de "*a ballerina*" e suas paráfrases, incluindo os *bsems* dos vértices. Note que criamos vértices distintos

para diferentes categorias gramaticais de uma mesma palavra, "*female*". A relevância desta medida será esclarecida adiante.

Combatendo a sobre-redundância

Empregamos duas restrições a fim de evitar a sobre-redundância. A primeira é completa porque sua semântica declarativa já sintetiza tudo o que queremos dizer em relação a essa questão, ao passo que a outra é redundante, incompleta, porém fornecida para aumentar a propagação. A restrição completa é a seguintes:

```
defprinciple "principle.over-redundancy" {
   dims {PA DPA}
constraints {
   forall V: forall U:
      (forall P: P in V.PA.attr.sem =>
         (P in U.PA.attr.bsem |
         (exists W: W ~= V & edge(U W DPA) &
            P in W.PA.attr.sem)))
      =>
      ~edge(U V DPA)
}
```

(VI.8)

A restrição consiste em proibir que um vértice *v* seja filho de *u* caso, para todo literal *p* do conteúdo semântico atual de *v*, *p* possa ser encontrado ou em $bsem(u)$ ou no conteúdo semântico de algum filho *w* conhecido de *u* tal que não seja *v*. Em outras palavras, cada filho de *u* deve fazer diferença. Entretanto, interessa-nos a forma otimizada de (VI.8) abaixo, que propicia maior propagação:

$$hds(u,v) = daughters(u,dpa) - \{v\}:$$
$$\forall v,u \left(sem(v) \subseteq bsem(u) \cup \bigcup_{hds(u,v)} \left(sem(1),...,sem(|\mathbb{V}|) \right) \Rightarrow \neg u \rightarrow_{dpa} v \right) \quad \text{(VI.9)}$$

A restrição (VI.9) é especialmente ativa após algumas escolhas terem sido feitas. Suponhamos, em nosso exemplo de "*a ballerina*", que "*dancing*" seja a única palavra selecionada até

o momento para a lexicalização. Sejam *u* e *v* os vértices correspondentes às palavras "*a*" e "*dancing*", respectivamente. Nesse caso, existe uma aresta de *u* para *w* em DPA em decorrência dos demais princípios, isto é, *w* é um filho conhecido de *u*, e o conseqüente da implicação em (VI.9) é falso. Isso, aliado ao fato de que $bsem(u) = \varnothing$ e $bsem(v) = \{dance(x)\}$, impõe a seguinte restrição:

$$\{dance(x)\} \not\subseteq \bigcup_{hds(u,v,dpa)} \left(sem(1),...,sem(|\mathbb{V}|)\right) \qquad \text{(VI.10)}$$

Segue que nenhum outro filho de *u* pode implicar $dance(x)$. Como isso não vale para os vértices correspondentes a "*ballerina*" e "*dancer*", eles são eliminados como potenciais filhos de *u* e, então, deletados por falta de potenciais pais. Alternativamente, se "*ballerina*" tivesse sido selecionada primeiro, (VI.9) detectaria trivialmente a sobre-redundância de todas as demais palavras e analogamente implicaria sua deleção.

Por outro lado, a restrição redundante assegura que, para todo par de vértices *u* e *v*, se o conteúdo semântico atual de *v* está contido no de *u*, então eles jamais podem ser *irmãos*. Formalmente:

$$\forall u,v\left(sem(u) \subseteq sem(v) \Rightarrow \neg u \rightarrow_{dpasis} v\right) \qquad \text{(VI.11)}$$

ou, em sua expressão PL:

```
defprinciple "principle.over-redundancy2" {
   dims {PA DPA DPASIS}
constraints {
   forall V: forall U:
      (forall P: P in V.PA.attr.sem =>
         P in U.PA.attr.sem)
      =>
      ~edge(V U DPASIS)
}
```

(VI.12)

A restrição (VI.11) é notável por ser ativa mesmo na ausência de escolha, visto que se estabelece entre irmãos em potencial, que usualmente têm seus *sems* suficientemente, se não completamente, determinados. Surpreendentemente, o principal efeito de (VI.11) é sobre a sintaxe, ao restringir alianças em $\mathbb{DS}$. Como instrumentamos o princípio **`climbing`** com sensibilidade a restrições entre irmãos, ele irá restringir todo vértice em $\mathbb{PA}$ a ter como pai em $\mathbb{DS}$ o mesmo que tem em $\mathbb{PA}$ ou algum vértice pertencente a uma das suas árvores-irmãs em $\mathbb{PA}$. Em outros termos, quando (VI.11) detecta, por exemplo, que "*woman*" subsume "*female* (adj./n.)" e os restringe a não serem irmãos em $\mathbb{PA}$, **`climbing`** eliminará "*woman*" como um potencial pai em $\mathbb{DS}$ de "*female* (adj.)". Vale mencionar que uma vez que $\neg u \rightarrow_{dpasis} v$ é imposto, o princípio **`sisters`** implica $\neg v \rightarrow_{dpasis} u$.

Restrições redundantes de composicionalidade

Apesar de uma definição completa de composicionalidade semântica ser dada por (VI.5), introduzimos duas restrições redundantes a fim de promover maior propagação. A primeira delas pretende adiantar a detecção de vértices cuja contribuição semântica seja estritamente requerida mesmo antes de os atributos *sem* de seus pais tornarem-se suficientemente determinados. Isso ocorre por meio de uma estratégia análoga àquela de (VI.8), levantando, para todo vértice *v*, a hipótese de como seria o conteúdo semântico *total* se *v* fosse deletado, da seguinte forma:

$$\boxed{\begin{array}{l} hdw(u,v) = down(u,dpa) - \{v\} \\ htotsem(v) = \bigcup_{hdw(1,v)} \left(sem(1),\dots,sem\left(|\mathbb{V}|\right)\right) \end{array}} : \qquad \text{(VI.13)}$$

$$\forall v \left(1 \overset{del}{\rightarrow}_{pa} v \Rightarrow sem(v) \subseteq htotsem(v) \right)$$

onde 1 identifica a raiz da análise. Infelizmente, (VI.13) não é muito útil em nosso exemplo corrente, aplicando-se melhor a casos em que há maior número de vértices internos alternati-

vos. Por exemplo, na lexicalização de (VI.2), esta restrição foi imediatamente capaz de inferir que "*lovely*" não deveria ser deletado, por ser o único vértice contribuindo $lovely(y)$.

A segunda restrição redundante de composicionalidade pretende adiantar a detecção de vértices não contando com irmãos suficientes para realizar o conteúdo semântico atual de seus pais. Para tanto, a seguinte restrição é imposta:

$$\boxed{\begin{array}{l} ms(v) = mothers(v, dpa) \\ eqsis(v) = \begin{cases} \varnothing & (v \text{ deletado em } pa) \\ daughters(v, dpasis) \cup \{v\} & (\text{caso contrário}) \end{cases} \end{array}} : \\ \forall v \left(\begin{array}{c} \bigcup_{ms(v)} \left(sem(1), \ldots, sem\left(|\mathbb{V}|\right) \right) \\ = \\ \bigcup_{ms(v)} \left(bsem(1), \ldots, bsem\left(|\mathbb{V}|\right) \right) \cup \bigcup_{eqsis(v)} \left(sem(1), \ldots, sem\left(|\mathbb{V}|\right) \right) \end{array} \right) \quad \text{(VI.14)}$$

que se lê: "o conteúdo semântico atual dos pais de um vértice é a igual ao conteúdo de sua semântica de base mais o conteúdo semântico atual deste vértice e o de seus irmãos". Vale observar que, quando v é deletado, tanto $ms(v)$ quanto $eqsis(v)$ tornam-se vazios, de modo que (VI.14) ainda é válido. Essa restrição é especialmente interessante porque nossas novas versões dos princípios **`climbing`** e **`barriers`**, que interfaceiam $\mathbb{DS}$ e $\mathbb{PA}$, propagam restrições de irmandade em todas as direções. Em associação com (VI.12) e (VI.14), estes princípios promovem uma interessante interação entre sintaxe e semântica. Retomando nosso exemplo, seja v o vértice "*female* (n.)". Antes que qualquer seleção seja realizada, a restrição (VI.12) infere que apenas "*dancing*", "*person*" e "*dancer*" podem ser irmãos de v em $\mathbb{PA}$ e, portanto, (agora devido a **`climbing`**) filhos de v em $\mathbb{DS}$. Eles não podem ser pais de v porque a valência deste em $\mathbb{DS}$ e o princípio **`climbing`** são suficientes para estabelecer que, se v tem qualquer pai em toda a $\mathbb{DS}$, este é "*a*". Novamente considerando a valência de v, é possível inferir que, se v tem qualquer irmão em toda a $\mathbb{DS}$, este é "*dancing*", ou seja, o único adjetivo no conjunto original de candidatos a irmãos. Desta vez, é **`barriers`** que propaga esta nova

informação de volta à $\mathbb{PA}$. Este princípio agora "sabe" que os irmãos de *v* em $\mathbb{PA}$ devem vir também de (i) a árvore abaixo de *v* em $\mathbb{DS}$, (ii) uma de suas árvores-irmãs em $\mathbb{DS}$ ou (iii) alguma árvore $\mathbb{DS}$ cuja raiz pertença a $eqsis(inter)$ para algum vértice *inter* que aparece – em $\mathbb{DS}$ – entre *v* e um de seus pais em $\mathbb{PA}$. Em nosso exemplo, sabe-se que (ii) e (iii) são conjuntos vazios, ao passo que (i) é, no mais, "*dancing*". Conseqüentemente, "*dancing*" é o único irmão potencial de *v* em $\mathbb{PA}$. Agora, (VI.14) é finalmente capaz de colaborar. Como "*a*" é o único pai em $\mathbb{DS}$ de *v*, e todo artigo tem semântica de base vazia, pode-se equacionar:

$$\bigcup_{ms(v)} \left(sem(1), \ldots, sem\left(|\mathbb{V}|\right)\right) = \bigcup_{eqsis(v)} \left(sem(1), \ldots, sem\left(|\mathbb{V}|\right)\right) \qquad \text{(VI.15)}$$

Muito embora não se saiba se *v* chegará a ter pais ou irmãos, (VI.14) sabe que o lado esquerdo da equação gera toda a semântica pretendida ou nada, ao passo que o lado direito produz também nada ou, no máximo, $dance(x) \wedge female(x)$. Sendo assim, a única solução para a equação é nada em ambos lados, implicando que $eqsis(v)$ é vazio e, assim, *v* é deletado por definição.

Tal forte interação é apenas possível porque criamos vértices distintos para as diferentes categorias gramaticais – ou melhor, as duas valências diferentes em $\mathbb{DS}$ – de "*female*". Com restrições mais pesadas e complicadas, seria possível ter a mesma propagação para um único vértice selecionando entre diferentes categorias gramaticais. Entretanto, isso não parece valer o esforço, pois um algoritmo de criação de modelo seria perfeitamente capaz de detectar as valências divergentes de $\mathbb{DS}$, criar tantos vértices quantos fossem necessários e distribuir as entradas lexicais corretas entre os mesmos.

VI.2 Desempenho e considerações finais

As idéias apresentadas acima foram implementadas em sua totalidade sobre a plataforma XDK. Entretanto, as otimizações de seleção, extremamente importantes para a propagação, foram realizadas manualmente, já que o QO ainda não é capaz de realizá-las. Alguns experimentos de prova de conceito foram realizados com entrada similar a (VI.2), ou seja, e lingüística e combinatoriamente análoga ao exemplo "*a ballerina tapped a lovely she-dog*". Neles, o sistema foi capaz de gerar todas as paráfrases sem qualquer falha (*backtracking*) na busca, o que implica dizer que a propagação foi máxima, não havendo uma má escolha em momento algum do processo. Esse é um resultado bastante animador, que demonstra o potencial da XDG em fazer diferentes dimensões de descrição lingüística colaborar para a redução da complexidade. Experimentos com constructos lingüísticos mais complexos, tais como orações relativas e sintagmas preposicionais, não foram realizados pois identificamos a necessidade de modificar substancialmente o tratamento que lhes tem sido dado na prática corrente da XDG, com vistas a incorporar outras dimensões de descrição lingüística relevantes à RL.

VII Considerações Finais e Trabalhos Futuros

Esperamos ter demonstrado, ao longo deste trabalho, o potencial da plataforma XDG/XDK na RL. Estamos cientes de que existem inúmeras questões em aberto entre o que foi ora apresentado e o que seria um sistema completo de RL. Por outro lado, dispomos de diversos projetos de resposta a algumas dessas questões, que, entretanto, não tivemos oportunidade de abordar neste trabalho, seja por serem ainda muito preliminares, seja por falta de tempo e espaço. Por exemplo, perdemos muito tempo tentando incorporar a um modelo de geração baseado na XDG um módulo completo de Geração de Expressões Referenciais. Hoje cremos que tal empreitada é, a um só tempo, impossível e desnecessária. Ainda assim, vemos muita vantagem e chances de sucesso em um tal sistema que realize apenas a porção da GRE estritamente necessária para construir esqueletos de cadeias de co-referência, sem a preocupação, por exemplo, de gerar anáforas nominais completas, mas capaz de decidir entre a necessidade desse tipo de dispositivo referencial ou a aplicação de outros, como pronomes, que poderiam ser tratados em sua generalidade por serem palavras de classe fechada.

Cremos não ser impossível desenvolver com a XDG uma "gramática/teoria de texto" abrangendo diversas dimensões (sintáticas, semânticas e discursivas) e conciliando, de forma mais ou menos direta, elementos de teorias diversas, tais como X-barra (especialmente no tocante à ligação), Teoria de Centering, Teoria de Estruturação Retórica e Teoria das Veias. Na verdade, já iniciamos esforços nesse sentido, mas a grande dúvida recai sobre se conseguiríamos obter propagação suficiente.

Sem dúvida, valerá a pena prosseguir o trabalho com o QO e explorar os meandros das EQs de seleção, que, sem dúvida, ainda guardam surpresas. Outra possibilidade interessante seria incorporar a um futuro sistema de geração modelos probabilísticos de língua, algo que já foi realizado com sucesso para a aplicação de *parsing*.

Referências bibliográficas

(Copestake et al., 2001) Copestake, A. A.; Lascarides, A.; Flickinger, D. (2001). An algebra for semantic construction in constraint-based grammars. *In Meeting of the Association for Computational Linguistics*, 132-139, 2001.

(Bateman et al., 1998) Bateman, J.; Kamps, T.; Kleinz, J.; Reichenberger, K. Communicative goal-driven nl generation and data-driven graphics generation: an architectural synthesis for multimedia page generation. *In Proceedings of the 9th International Workshop on Natural Language Generation*, Ontario, Canada, 1998.

(Bresnan & Kaplan, 1982) Bresnan, J.; Kaplan, R. Lexical-Functional Grammar: A Formal System for Grammatical Representation. *In* Joan Bresnan, ed., *The Mental Representation of Grammatical Relations*, 173-281, The MIT Press, 1982.

(Brito, 1995) Brito, L. F. *Por uma Gramática de Línguas de Sinais*. Tempo Brasileiro Ed., Departamento de Lingüística e Filologia, Universidade Federal do Rio de Janeiro, 1995

(Danlos, 1987) Danlos, L. *The Linguistic Basis of Text Generation.* Cambridge University Press, 1987.

(Debusmann, 2004) Debusmann, R. Multiword Expressions as Dependency Subgraphs. *In Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics* (*ACL-2004, "Multiword Expressions: Integrating Processing" Workshop)*, Barcelona, 2004.

(Debusmann, 2006) Debusmann, R. *Extensible Dependency Grammar: A Modular Grammar Formalism Based on Multigraph Description*. Tese de doutorado, Naturwissenschaftlich-Technischen Fakultäten der Universität des Saarlandes, 2006.

(Debusmann, 2007) Debusmann, R. Scrambling as the Combination of Relaxed Context-Free Grammars in a Model-Theoretic Grammar Formalism. *In Proceedings of the ESSLLI 2007 Workshop Model-Theoretic Syntax at 10*, Dublin, 2007.

(Debusmann & Kuhlmann, 2007) Debusmann, R.; Kuhlmann, M. *Dependency Grammar: Classification and Exploration*, Project Report (CHORUS, SFB 378), Programming Systems Lab, Saarland University, 2007.

(Debusmann et al., 2005) Debusmann, R.; Postolache, O.; Traat, M. A Modular Account of Information Structure in Extensible Dependency Grammar. *In Sixth International Conference on Intelligent Text Processing and Computational Linguistics* (*CICLING-2005*), México, 2005.

(Debusmann et al., 2004a) Debusmann, R; Duchier, D; Kruijff, G. J. Extensible Dependency Grammar: A New Methodology. *In Proceedings of the 20th International Conference on Computational Linguistics (COLING-2004, Workshop on Recent Advances in Dependency Grammar)*, Geneva, 2004.

(Debusmann et al., 2004b) Debusmann, R; Duchier; Koller, A; Kuhlmann, M.; Smolka, G.; Thater, S. A Relational Syntax-Semantics Interface Based on Dependency Grammar. *In Proceedings of the 20th International Conference on Computational Linguistics (COLING-2004)*, Geneva, 2004.

(Debusmann et al., 2004c) Debusmann, R.; Duchier, D.; Kuhlmann, M. Multidimensional Graph Configuration for Natural Language Processing. *In Proceedings of the International Workshop on Constraint Solving and Language Processing*, 59-73, Roskilde, Denmark, 2004.

(Duchier, 2002) Duchier, D. Configuration of labeled trees under lexicalized constraints and principles, *Journal of Language and Computation*, 2002.

(Duchier, 1999) Duchier, D. Axiomatizing dependency parsing using set constraints. *In Proceedings of the 6th Meeting on the Mathematics of Language*, USA, 1999.

(Duchier & Debusmann, 2001) Duchier, D.; Debusmann, R. Topological dependency trees: A constraint-based account of linear precedence. *In Proceedings of the 39th ACL*, France, 2001.

(Duchier & Thater, 1999) Duchier, D.; Thater, S. Parsing with Tree Descriptions: a Constraint-Based Approach. *In Sixth International Workshop on Natural Language Understanding and Logic Programming (NLULP'99)*, 17-32, Las Cruces, New Mexico, 1999.

(Duchier et al., 2002) Duchier, D.; Gardent, C.; and Niehren J. *Concurrent Constraint Programming in Oz for Natural Language Generation* (disponível em http://www.ps.uni-sb.de/~niehren/Web/Vorlesungen/Oz-NL-SS01/vorlesung e acessado em março de 2005), 2002.

(Eddy, 2002) Eddy, B. Toward balancing conciseness, readability and salience: an integrated architecture. *In Proceedings of the International Natural Language Generation Conference (INLG'02)*, New York, USA, 2002.

(Eddy et al., 2001) Eddy, B.; Bental, D.; Cawsey, A. An algorithm for efficiently generating summary paragraphs using tree-adjoining grammar. *In Proceedings of EWNLG01*, 8th European Workshop on Natural Language Generation, Toulouse, France, julho de 2001.

(Gardent, 2002) Gardent, C. Generating Minimal Definite Descriptions. *In Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL'02)*, 2002.

(Gardent & Thater, 2001) Gardent, C.; Thater, S. Generating with a Grammar Based on Tree Descriptions: a Constraint-Based Approach. *In* Bird, S., ed., *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics (ACL'01)*, 2001.

(Kibble & Power, 2004) Kibble, R.; Power, R. Optimizing Referential Coherence in Text Generation. *Computational Linguistics*, 30 (4), 401-416, 2004.

(Koller & Striegnitz, 2002) Koller, A.; Striegnitz, K. Generation as Dependency Parsing. *In Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2002, 17-24.

(Lytinen, 1992) Lytinen, S. L. Conceptual Dependency and its Descendants. *Computers and Mathematics with Applications*, 23(2-5), 51-73.

(Mann & Thompson, 1988) Mann, W.; Thompson, S. Rhetorical Structure Theory: towards a Functional Theory of Text Organization. *Text*, 8 (3), 243-281, 1988.

(Martins, 2004) Martins, R. T. *A Língua Nova do Imperador*, dissertação de doutorado, Instituto de Estudos da Linguagem, Universidade Estadual de Campinas (UNICAMP), 2004.

(Martins et al., 2002) Martins, R. T.; Rino, L. H. M.; Nunes, M. G. V.; Oliveira Jr., O. N. The UNL distinctive features: evidences through a NL-UNL encoding task. *In Proceedings of the First International Workshop on UNL, other Interlinguas and their Applications*, LREC 2002, 08-13.

(Martins et al., 2000) Martins, R. T.; Rino, L. H. M.; Nunes, M. G. V.; Montilha, G.; Oliveira Jr., O. N. An interlingua aiming at communication on the Web: how language-independent can it be? *In Proceedings of the Workshop on Applied Interlinguas: Practical Applications of Interlingual Approaches to NLP*, ANLP-NAACL 2000.

(McKeown et al., 1998) McKeown, K. R.; Jordan, D. A.; Hatzivassiloglou, V. 1998. Generating patient specific summaries of online literature. *In AAAI Spring Symposium on Intelligent Text Summarisation*, 34-43, 1998.

(Meteer, 1992) Meteer, M. *Expressibility and the Problem of Efficient Text Planning.* Pinter, London, 1992.

(Pelizzoni & Nunes, 2005) Pelizzoni, J. M.; Nunes M. G. V. Flexibility, Configurability and Optimality in UNL Deconversion via Multiparadigm Programming. *In Research on Computing Science*, 12, 175-194, México, 2005.

(Pollard & Sag, 1994) Pollard, C.; Sag, I. A. *Head-Driven Phrase Structure Grammar*. University of Chicago Press, 1994.

(Power, 2000) Power, R. Planning Texts by Constraint Satisfaction. *In Proceedings of the 17th conference on Computational Linguistics*, 642-648, Saarbrücken, Alemanha, 2000.

(Power et al., 2003) Power, R.; Scott, D.; Bouayad-Agha, N. Document structure. *Computational Linguistics*, 29(2), 211-260, 2003.

(Pullum & Scholz, 2001) Pullum, G. K.; Scholz, B. C. On the distinction between model-theoretic and generative-enumerative syntactic frameworks. *In* de Groote, P.; Morrill, G.; Retoré, C. (eds.) *Logical Aspect of Computational Linguistics: 4th International Conference*, Lecture Notes in Artificial Intelligence, Springer, 17-43, 2001.

(Reape & Mellish, 1999) Reape, M.; Mellish, C. Just what is aggregation anyway? *In European Workshop on Natural Language Generation*, Toulouse, France, maio de 1999.

(Reiter & Dale, 2000) Reiter, E.; Dale, R. *Building Natural Language Generation Systems*, Cambridge University Press, 2000.

(Rich & Knight, 1991) Rich, E.; Knight, K. *Artificial Intelligence*, 2nd edition, McGraw-Hill, 1991.

(Robin & McKeown, 1996) Robin, J.; McKeown, K. R. Empirically designing and evaluating a new revision-based model for summary generation. *Artificial Intelligence*, 85(1-2), agosto de 1996.

(Rudell, 1986) Rudell, R. L. *Multiple-Valued Logic Minimization for PLA Synthesis*, Technical Report (M86/65), EECS, University of California at Berkeley, 1986.

(Schank & Rieger, 1974) Schank, R. C.; Rieger, C. J., III. Inference and the Computer Understanding of Natural Language. *Artificial Intelligence*, 5, 373-412, 1974.

(Schulte, 2002) Schulte, C. *Programming Constraint Services: High-Level Programming of Standard and New Constraint Services*, Lecture Notes in Computer Science Series, Springer-Verlag, 2002.

(Steedman, 2000) Steedman, M. Information structure and the syntax-phonology interface. *Linguistic Inquiry*, 31(4), 649-689, 2000.

(Stone & Doran, 1997) Stone, M.; Doran, C. Sentence planning as description using tree-adjoining grammar. *In Proceedings of the Assosciation for Computational Linguistics*, 1997, 198-205.

(Russel & Norvig, 1995) Russell, S.; Norvig, P. *Artficial Intelligence: A Modern Approach*, Prentice-Hall, 1995.

(Wilkinson, 1995) Wilkinson, J. *Aggregation in Natural Language Generation: Another Look,* technical report, Computer Science Dept., University of Waterloo, 1995.

(Williams, 2003) Williams, S. Language Choice Models for Microplanning and Readability. *In Proceedings of the Student Workshop of the Human Language Technology and North*

American Chapter of the Association for Computational Linguistics Conference (HLT-NAACL03 Student Workshop), 13-18, 2003.

(Van Roy & Haridi, 2004) Van Roy, P.; Haridi, S. *Concepts, Techniques, and Models of Computer Programming*, MIT Press, 2004.

(Zock, 1987) Zock, M. Proposal for simulating on-line generation by giving priority either to lexical choices or syntactic structure. *In COGNITIVA.*, 1987.

Índice Remissivo

#

$\forall!$, **98**, ***101***

A

agregação, **8**

alçamento

~ de expressões livres, **102**

allbutone, **98**

análise

tipo de ~, **62**

aprofundamento de quantificadores, ***97***

arboreidade, **39**

arquiteturas integradas, **7**

árvore

~ eletrostática, **35**

árvore eletrostática

descrição de ~, **35**

atributo

valencial, **60**

atributos, **58**

Avaliador, **86**

axiom, ***128***

B

biunivocidade entre descritores e referentes, **8**

bsem, ***127***

C

classe lexical, **72**

CNF, **107**

cobertura, 3

colaboração, 22

completude

~ semântica, **127**

complexidade, **3**

combinatória, 16

compsem, ***108***

comunicação

multidirecional, 21

concentrador, **128**

conjunção, **75**

lexical, **75**

operador de ~, **75**

criação de modelo, **26**

CSE, **113**

CSP, **27**

D

DAE, **35**

deconversão UNL, **41**

defattrstype, **64**

defentrytype, **65**

deflabeltype, **63**

deleção, **58**

destino, **50**

DG, **38**

dimensão, **50**

metáfora, **53**

disjunção, **74**

lexical, **74**

operador de ~, **74**

distribuição, **22**

estratégia de ~, **23**

first-fail, **24**

DNF, **106**

dom(v), **15**

dom_0(v), **15**

domínio, **15**

inicial, **15**

E

eletrostática

árvore ~, **35**

eliminação

~ de subexpressões comuns, **113**

~ de variável, **95**

eliminação de quantificadores. *Consulte* EQ

~ parcial, **93**

eliminação de simetrias, **25**

eliminação de variáveis, **93**

entry, **42**

EQ, **93**

~ de seleção, **105**

básica, **97**

espúria

~ variável, **116**

estabilidade consistente, **23**

estratégia de distribuição, **23**, *vide* distribuição

estrutura textual, **8**

expressão referencial, 9

expressão trivial, **114**

expressividade, 3

exprimibilidade, 3

extensibilidade, **9**

F

FinishUp, **44**

first-fail, **24**

flat, **30**

Forma Conjuntiva Normal, **107**

Forma Disjuntiva Normal, ***106***

G

Gamma, **44**

generalidade, **2**

geração de expressões referenciais, 9

gramática

XDG, **62**

Gramática

~ de Adjunção de Árvores, **34**

~ de Dependência, **38**

~ de Descrições Arbóreas, **35**

I

inferência

multidirecional, 21

instanciabilidade, **2**

iset, **65**

L

let, **114**

lexicalização, **5**, **60**

léxico

conjunção, **75**

disjunção, **74**

sincronização lexical, **77**

limite

inferior, **89**

superior, **89**

linkAboveBelow1or2Start, 60

linkBelow1or2Start, 60

localidade

de inferência, **23**

M

Manati, **41**

Mapper, **43**

matriz atributo-valor, **59**

medida de qualidade, 6

metáfora multidimensional, **53**

modelagem, **15**

modelo, **15**

criação de ~, **26**

modularidade, **6**

monossentencial, **7**

monotonicidade, 21

movimento, **57**

Mozart. *vide* Oz, www.mozart-oz.org

multigrafos, **11**

O

oráculo, **44**

origem, **50**

otimalidade, **6**

otimização

~ de código, **85**

otimizador

~ de código, **85**

Oz, 19, 25, www.mozart-oz.org

P

paradigma concorrente de restrições, **15**, *vide* restrição

PCR, **10**, **15**

PL, ***79***

~ arbórea, **86**

PLA, **86**

~ Estendida, **86**

PLAE, **86**

plurissentencial, **7**

POS, **43**

Precond, **44**

princípio, **66**

acepção 1, **67**

acepção 2, **67**

Principle Language, ***79***

Principle Writer, **79**

Problemas de Satisfação de Restrições, **27**

projeção, **52**

projetividade, **57**

propagação (de restrições), **18**

propagador, **19**

PSRs, **27**

PW, **79**

Q

QO, **85**

QuadOptimizer, ***85***

qualidade

medida de ~, 6

quantificadores, **39**

R

raiz da análise, **58**

Recapitulando, ***121***

reificação, **89**

relação, **50**

resolução

por busca, **17**

restrição, **15**

~ de seleção, **105**

~ reificada, **89**

básica, **15**

de distribuição, **23**

não-básica, **18**

redundante, **26**

restrições

sobre conjuntos finitos, **88**

RST, **30**

S

satisfação lexical, **76**

SDNF, **110**

semântica

- declarativa, **18**
- operacional, **18**

simetria, **25**

- eliminação de ~, 25

sincronização lexical, **77**

sinergia, **6**, 22

sobre-redundância, **121**

T

TAG, **34**

TDG, **35**

Tipador, **86**

tipo de análises, ***63***

topmothers, ***112***

Tree Adjoining Grammars, **34**

trivial

- expressão ~, **114**

U

UL, **62**

UNL, **41**

- deconversão, **41**

User Language, ***62***

UW, **41**

V

valência, **60**

valency, **65**

variáveis gráficas, **91**

variável espúria, **116**

VDAE, **36**

vw, ***58***

X

XDG, 10

- gramática, **62**
- introdução à, **49**
- princípio, **66**

XDK, **11**

- introdução ao, **49**
- Principle Writer, **79**